# AN12185

## A71CH Host software porting guidelines

Rev. 1.0 — 09 July 2018
492410

Application note
COMPANY PUBLIC

**Document information**

| Info | Content |
|---|---|
| **Keywords** | Secure Element, A71CH, porting, guide |
| **Abstract** | This document provides a detailed guide for porting A71CH Host Library to different types of platforms to work with the A71CH security IC |

### Revision history

| Rev | Date | Description |
|---|---|---|
| 1.0 | 20180709 | First release |

## Contact information

For more information, please visit: http://www.nxp.com

# 1. Introduction

This document explains how to port the A71CH Host Library to other platforms to work with the A71CH security IC. It gives an overview of the files of the A71CH Host Library that require being modified and it provides step-by-step instructions on how to modify them to port the library as well as illustrated examples of this process.

# 2. A71CH overview

The A71CH is a ready-to-use solution, enabling ease-of-use security for IoT device makers. It is a secure element capable of securely storing and provisioning credentials, securely connecting IoT devices to public or private clouds and performing cryptographic device authentication.

The A71CH solution provides basic security measures protecting the IC against many physical and logical attacks. It can be integrated with various host platforms and operating systems to secure a broad range of applications. In addition, it is complemented by a comprehensive product support package, offering easy design-in with plug & play host application code, easy-to-use development kits, documentation and IC samples for product evaluation.

# 3. A71CH Host Library architecture

The A71CH Host Library translates function calls into APDUs that are transferred through an I²C interface to the A71CH security IC. The A71CH executes the different APDUs and gives back the results to the A71CH Host Library through the same interface. The complete set of A71CH Host Library functions can be called from communication stacks like TLS or an application running on the host. Therefore, the A71CH Host Library behaves as the interface between a host microcontroller application and the A71CH security IC. Fig 1 depicts the complete A71CH Host Library architecture.

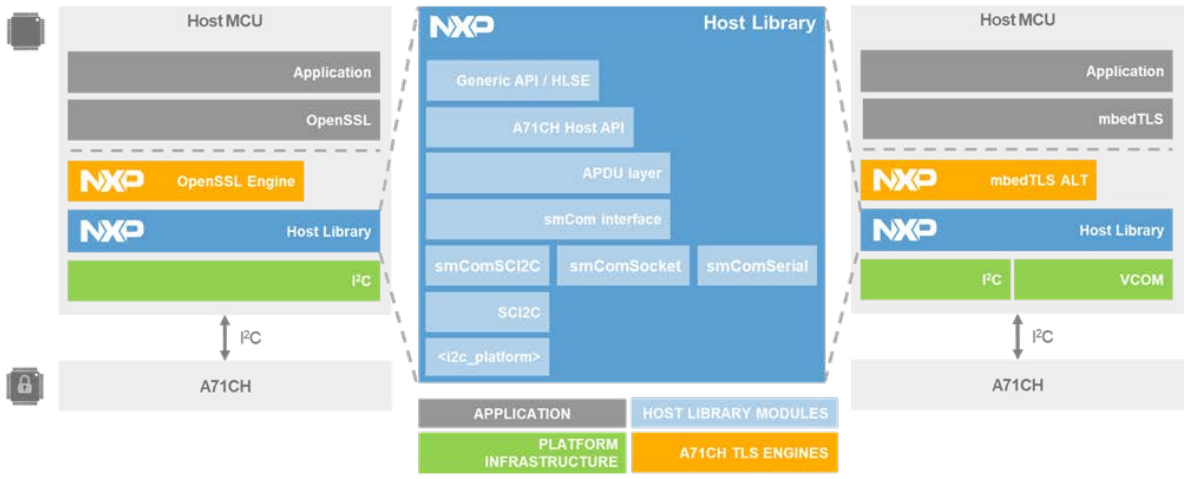


**Fig 1. A71CH Host Library architecture**

Details on the folder distribution of the A71CH Host Library can be found in [AN_A71CH_HOST_SW].

## 3.1 Platform drivers

In Fig 1, it can be observed that the <i2c_platform> layer acts as a link between the SCI2C protocol and I²C platform infrastructure. It is platform specific; i.e., it must be adapted depending on the platform.

The A71CH Host library already provides the <i2c_platform> layer adapted for i.MX6UltraLite and Kinetis MCU (K64F). In addition, it can be run in other platforms if this layer is modified accordingly.

The SCI2C is a communication protocol based on an I²C physical interface and data link layer, a SMBus-based network layer and bus protocol as well as a mapping layer to convey ISO/IEC 7816-4 based communication [SCI2C].

The SMBus (System Management Bus) is a two-wire interface through which various system component chips can communicate with each other and with the rest of the system. SMBus provides a control bus for the system to pass messages to and from devices instead of using individual control lines, helping to reduce pin count and system wires. It is derived from I²C and is therefore based on the same principles of operation. For this reason, it can be defined on top of an I²C physical interface to enforce arrangement of data packets. Further information on the SMBus can be found in [SMBUS] [AN_SMBUS].

This guide assumes that the host MCU platform supports “Repeated Start” and “Block Read” and it gives guidelines on how to modify the required files to port the A71CH Host Library. If these features are not supported in the target platform, jump to section 4.1.1.

The files to be modified are the following:

- **i2c_<platform>.c**: Platform specific I²C code; the I²C API used by SCI2C library is defined. The file i2c_a7.c is in the /hostLib/platform/imx/ directory and the file i2c_kinetis.c is in the /hostLib/platform/kinetis/ directory.
- **sm_timer.c**: This file defines the sleep functionality. It must be implemented according to the timers of the target platform. Located in the folder /hostLib/platform/generic/.
- **timer_kinetis_<platform>.c**: These files define the implementation of the sleep functions for different platforms. Located in the /hostLib/platform/kinetis directory.
- **sm_printf.c**: Printf implementation. This is usually platform independent but in case the target platform doesn’t support standard libraries or has special conditions, it needs to be adapted. Located in the /hostLib/platform/generic directory.

Fig 2 shows the distribution of the files mentioned above. In the following sections, information about implementing and adapting these files is given.

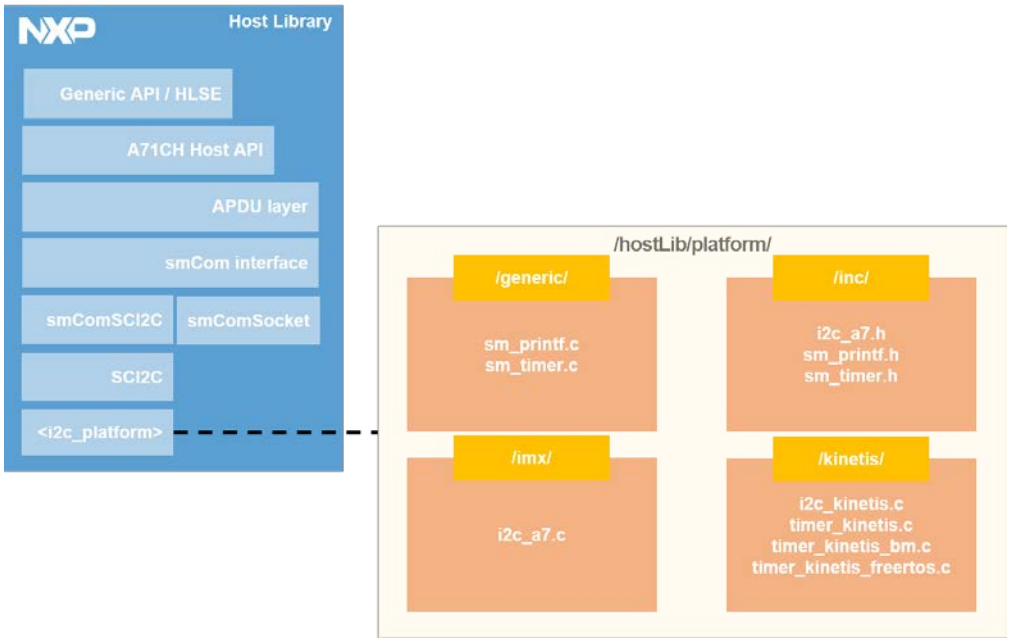


**Fig 2. Folder structure and location of each of the files.**

## 4. Porting the library

To port the A71CH Host library to other platforms, the user will first need to make sure if the aimed platform features an I²C-dedicated hardware, or if a bit-banging mechanism is required. Additionally, the host platform's I²C driver must support the SMBus features "Block Read" and "Repeated Start" to ensure compatibility with SCI2C protocol.

The library is written in pure 'C' code that is platform independent and should compile on any system, with or without OS. However, the user must adapt some files, as mentioned in section 3.1; these files are the I²C platform specific code, the timers according to the platform and optionally, the sm_printf implementation. The last one is usually platform independent but there may be special target platforms that don't support it and need to be adapted.

In this document, two scenarios to port the A71CH Host Library are described: porting to a Linux environment, using the i.MX6UltraLite board as an example, and porting to an RTOS or bare metal environment, using the Kinetis MCU K64F as an example.

The SCI2C protocol used by the A71CH security IC is the revision 1.5 or later (not compatible with version 2.x). As mentioned in section 3.1, to ensure compatibility with SCI2C, the host platform's I²C driver must support:

- **Block Read**
- **Repeated Start**

## 4.1 SMBus Block Read and Repeated Start

Block Read is an SMBus packet implemented in SCI2C protocol. The main feature of Block Read packets is that the response length is encoded in the first byte of the response as shown in Fig 3. The first response of the slave device (color gray) is the “A”, which calls for ACK. Then, the “LEN” byte indicates the total length of the data that will be sent from the slave to the master device.

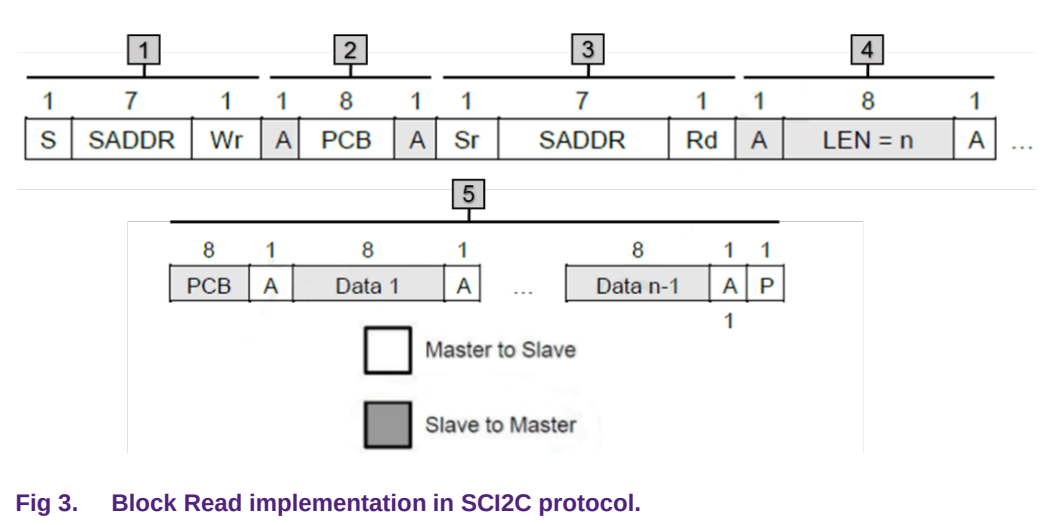


**Fig 3. Block Read implementation in SCI2C protocol.**

Another feature of the Block Read is that it uses Repeated Start. Repeated Start can be recognized with the “Sr” box in Fig 3. It is used to continue transmission with the same slave device in the opposite direction, thus allowing bidirectional W/R operations between master and slaves using the same uninterrupted channel. The communication flow of the Block Read in Fig 3 is:

1. The master device sends the first command “S – SADDR – Wr”; where “S” means Start condition, “SADDR” is the slave address and “Wr” means Write bit (0).
2. The slave device acknowledges, and the master sends the protocol control byte (“PCB”). This byte indicates the operation that is going to be performed. Finally, the slave device sends the “ACK” to the “PCB” byte.
3. Master sends the Repeated Start bit (“Sr”) with the slave address (“SADDR”) and a “Rd” bit, which means Read bit.
4. Slave acknowledges the Read Address. Then, the transmission is in the opposite way now: the slave is the one sending data and the master is waiting to read it. The slave sends the “LEN” packet which contains the number of bytes that will be sent. Then, the master device acknowledges each byte.
5. Finally, the slave starts sending the (return) PCB and optional data bytes. LEN equals the amount of data bytes returned plus one (the byte corresponding to the PCB). The Master must only send the STOP condition upon retrieving N bytes.

### 4.1.1 Block Read and Repeated Start features not supported

In most systems, data communication interfaces such as $I^2C$ bus or SMBus are handled by a dedicated hardware and the communication driver is already implemented:

parameters such as timing, levels and synchronization are managed by a dedicated circuitry. This dedicated hardware demodulates the signal and provides an already buffered data interface; thus, the MCU does not need to perform any software operation.

However, some MCUs do not have specific communication dedicated hardware. In this case, a technique called bit-banging is typically used. Bit-banging is a technique for implementing serial communications drivers using software instead of hardware. Software directly sets and samples the state of pins on the MCU and is responsible for all the signal parameters. For instance, I²C bus and SMBus drivers can be implemented using bit-banging over two available GPIO pins of the microcontroller.

# 5. Linux

## 5.1 Adapting I²C platform wrapper

The platform specific I²C code must be adapted to the target platform. These correspond to the layer that will convert the APDUs to binary code and send the data to the A71CH security IC.

Read Block and Repeated Start features must be implemented to be compliant with SCI2C protocol [SCI2C]. Fig 4 illustrates the code in a Linux environment using the i.MX6UltraLite enabling the Repeated Start and Block Read features. The source code can be found in “platform/imx” folder, as is mentioned in section 3.1. By passing the two message structures via the packets structure as a parameter to the “`ioctl`” call ensures that Repeated Start is triggered.

```
1       messages[0].addr  = axSmDevice_addr;
2       messages[0].flags = 0;
3       messages[0].len   = txLen;
4       messages[0].buf   = pTx;
5       // NOTE:
6       // By setting the 'I2C_M_RECV_LEN' bit in 'messages[1].flags' one ensures
7       // the I2C Block Read feature is used.
8       messages[1].addr  = axSmDevice_addr;
9       messages[1].flags = I2C_M_RD | I2C_M_RECV_LEN;
10      messages[1].len   = 256;
11      messages[1].buf   = pRx;
12      messages[1].buf[0] = 1;
13      // NOTE:
14      // By passing the two message structures via the packets structure as
15      // a parameter to the ioctl call one ensures a Repeated Start is triggered.
16      packets.msgs      = messages;
17      packets.nmsgs     = 2;
18      r = ioctl(axSmDevice, I2C_RDWR, &packets);
```

**Fig 4. Implementation of Block Read and Repeated Start in Linux in “i2c_a7.c”.**

The code for the Block Read feature is highlighted in red. The usage of Block Read is ensured by setting the “I2C_M_RECV_LEN” bit in “messages [1]. flags”. The “len” and “buf [0]” field values of the messages [1] structure need special attention:

- “messages [1]. len” is interpreted by the I²C device driver as the size of the buffer (pRx) provided by the caller. This value must be at least 33, a condition fulfilled by providing a buffer of 256 byte.
- The value contained in “messages [1]. buf [0]” must be at least “1”. On the i.MX6UltraLite the value is “1”.

If both conditions are met, the I²C device driver will replace the value of “message [1]. len” by the value contained in “messages [1]. buf[0]”. Only then is the platform specific I²C bus driver invoked.

In addition, the developer needs to ensure that the packet size on the bus is set correctly during the initial SCI2C Parameter Exchange. This packet size can be set in “sci2c_cgf.h” file, located in “hostLib/libCommon/smCom” folder. However, the maximum amount of data bytes that can be fetched from the master to the slave in one block read is restricted to 31 data bytes.

## 5.2 Required header files to import

The location of the header to implement the I²C driver is inside “platform/inc” folder of the A71CH Host Library. The name of that file is “i2c_a7.h”. Fig 5 depicts the content of the file. Highlighted in red, are the different functions that must be implemented depending on the target platform.

```
1    #ifndef _I2C_A7_H
2    #define _I2C_A7_H
3
4    #include "sm_types.h"
5    "..."
6    typedef unsigned int i2c_error_t;
7    #define I2C_BUS_0   (0)
8
9    i2c_error_t axI2CInit( void );
10   void axI2CTerm(int mode);
11
12   #if defined(FREEDOM)
13   void axI2CResetBackoffDelay( void );
14   #endif /* FREEDOM */
15   i2c_error_t axI2CWriteByte(unsigned char bus, unsigned char addr, unsigned char * pTx);
16   i2c_error_t axI2CWrite(unsigned char bus, unsigned char addr, unsigned char * pTx, unsigned short
     txLen);
17   i2c_error_t axI2CRead(unsigned char bus, unsigned char addr, unsigned char * pRx, unsigned short
     rxLen);
18   i2c_error_t axI2CWriteRead(unsigned char bus, unsigned char addr, unsigned char * pTx, unsigned
     short txLen, unsigned char * pRx, unsigned short * pRxLen);
19
20   #endif // _I2C_A7_H
```

**Fig 5. Highlighted in red, functions of “i2c_a7.h” to implement**

The I²C driver implementation in Linux is layered. The top-level layer is contained in a file called “i2c-dev.c” and is referred to as I²C device driver. The bottom layer deals with the specific hardware of the I²C controller, thus being specific for the target platform. It is called I²C bus driver.

The I²C device driver is defined in “i2c-dev.c” and “i2c.c” files. These files represent an I²C adapter implementing IOCTL functions (system call for device specific input/output operations and other operations which cannot be expressed by regular system calls). Therefore, it is required to include them in the implementation of the i2c_a7.c driver. As can be observed in Fig 6, both “i2c_dev.h” and “i2c.h” header files have been included in “i2c_a7.c”.

```
#include "i2c_a7.h"
#include <stdio.h>
#include <string.h>

#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <linux/i2c-dev.h>
#include <linux/i2c.h>
#include <time.h>
```

**Fig 6. Included files in “i2c_a7.c” depending on the target platform**

In summary, the header “i2c_a7.h” provides the definitions of the functions to implement (highlighted in red), while “i2c_dev.h” and “i2c.h” header files provide the definitions and implementation of the Linux native I²C device driver (highlighted in blue). It is not necessary to implement the functions of the I²C device driver as they are implemented natively by the system.

The last part to be specified in “i2c_a7.c” is a reference to the specific I²C master (device node) the A71CH is connected to. To do this, assign the correct device node to the variable “devName”. This is depicted in Fig 7. Note that, in this case; i.e. when using the i.MX6UltraLite with the Arduino shield, the I²C interface used is “i2c-1” but depending on the manufacturer and the system specifications, the direction may be different (default values usually are “i2c-0” or “i2c-1”).

```
static int axSmDevice;
static int axSmDevice_addr = 0x48;      // 7-bit address
static char devName[] = "/dev/i2c-1";   // Change this when connecting to another host i2c master
port
```

**Fig 7. Declaration of I²C port**

## 5.3 Adapting timers

The timers are defined by the system and are used to manage interrupts. Fig 8 illustrates the example in this guide, using as target platform Linux, RTOS or bare metal. It is more straightforward to implement “sm_timer.c” in Linux or RTOS versions as those timers are

usually implemented natively. In a bare metal system, the timers are defined by the MCU manufacturer.

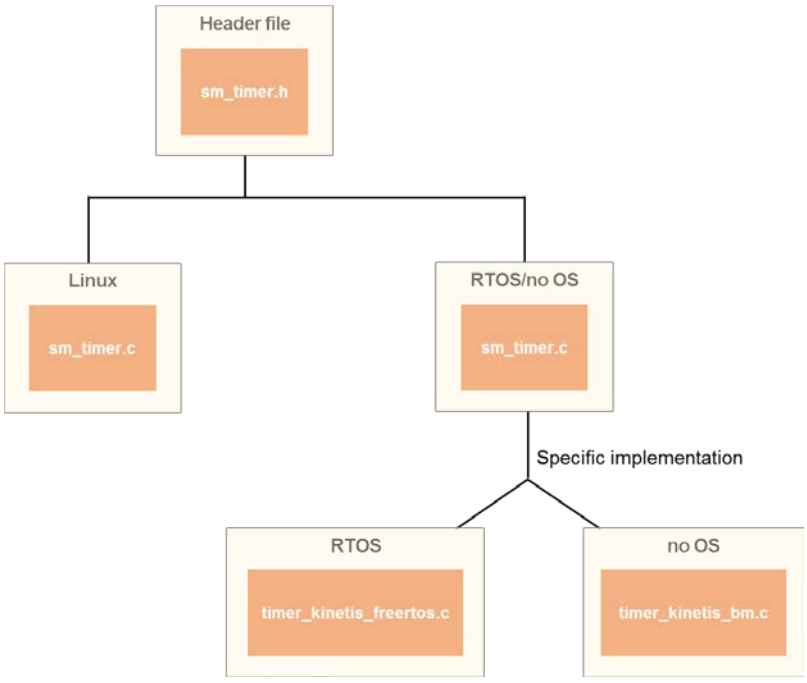


**Fig 8. Sm_timer.c implementation in each target platform**

In Linux, “sm_timer.h” defines three different functions to implement interruptions: “sm_initSleep” function to initialize the system tick counter, “sm_sleep” function to block the calling thread for a number of milliseconds and “sm_usleep” function to block the calling thread for `microsec` microseconds.

```
1    /* function used for delay loops */
2    uint32_t sm_initSleep(void);
3    void sm_sleep(uint32_t msec);
4    void sm_usleep(uint32_t microsec);
```

**Fig 9. Functions defined in “sm_timer.h”**

# 6. RTOS or bare metal

## 6.1 Adapting I²C platform wrapper

The platform specific I²C code must be adapted. This corresponds to the layer that will convert the APDUs to binary code and send the data to the A71CH security IC.

Read Block and Repeated Start features must be implemented to be compliant with SCI2C protocol. The source code can be found in “platform” folder, as mentioned in section 3.1.

Fig 10 illustrates the code for implementing Block Read and Repeated Start in a Kinetis MCU (Kinetis K64F). In this case, the Kinetis has its own API (“fsl_i2c.h” or “fsl_i2c_freertos.h”) that implements these features. The flag “kI2C_TransferRepeatedStartFlag” with a value of 0x2U (highlighted in blue), makes the transfer starts with a Repeated Start signal. The “I2C_MasterTransferBlocking” function (highlighted in red) performs a master polling transfer on the I²C bus.

```
masterXfer.slaveAddress = addr >> 1; // the address of the A70CM
masterXfer.direction = kI2C_Read;
masterXfer.subaddress = 0;
masterXfer.subaddressSize = 0;
masterXfer.data = pRx;
masterXfer.dataSize = 0; //We don't read anything here.
masterXfer.flags = kI2C_TransferRepeatedStartFlag;

result = I2C_MasterTransferBlocking(AX_I2CM, &masterXfer);
```

**Fig 10. Implementation of Block Read and Repeated Start in a Kinetis MCU -K64F- (either RTOS or bare metal) in “i2c_kinetis.c”**

## 6.2 Required header files to import

The location of the header to implement the I²C driver is inside “platform/inc” folder of the A71CH Host Library. The name of that file is “i2c_a7.h”. Fig 11 depicts the content of the file and shows the different functions that must be implemented.

```
#ifndef _I2C_A7_H
#define _I2C_A7_H

#include "sm_types.h"

#define SCI2C_T_CMDG 180 //!< Minimum delay between stop of Wakeup command and
start of subsequent command (Value in micro seconds)

#define I2C_IDLE              0
#define I2C_STARTED           1
#define I2C_RESTARTED         2
#define I2C_REPEATED_START    3
#define DATA_ACK              4
#define DATA_NACK             5
#define I2C_BUSY              6
#define I2C_NO_DATA           7
#define I2C_NACK_ON_ADDRESS   8
#define I2C_NACK_ON_DATA      9
#define I2C_ARBITRATION_LOST  10
#define I2C_TIME_OUT          11
#define I2C_OK                12
#define I2C_FAILED            13

```

```
23   typedef unsigned int i2c_error_t;
24   #define I2C_BUS_0   (0)
25
26   i2c_error_t axI2CInit( void );
27   void axI2CTerm(int mode);
28
29   #if defined(FREEDOM)
30   void axI2CResetBackoffDelay( void );
31   #endif /* FREEDOM */
32   i2c_error_t axI2CWriteByte(unsigned char bus, unsigned char addr, unsigned
     char * pTx);
33   i2c_error_t axI2CWrite(unsigned char bus, unsigned char addr, unsigned char *
     pTx, unsigned short txLen);
34   i2c_error_t axI2CRead(unsigned char bus, unsigned char addr, unsigned char *
     pRx, unsigned short rxLen);
35   i2c_error_t axI2CWriteRead(unsigned char bus, unsigned char addr, unsigned
     char * pTx, unsigned short txLen, unsigned char * pRx, unsigned short *
     pRxLen);
36
37   #endif // _I2C_A7_H
```

**Fig 11. Highlighted in red, functions of “i2c_a7.h” to implement**

A real-time operating system (RTOS) can manage and schedule events and manage interruptions; a wide use implementation of RTOS is FreeRTOS, used in this guide example.

Fig 12 shows the necessary files to include in the implementation of the “i2c_a7.c”. Again, the header file “i2c_a7.h” provides the definitions of the functions to implement (highlighted in red).

Two possible files, depending on the system configuration, are highlighted in blue. In case the system is compiled as a bare metal platform, “fsl_i2c.h” is used; on the other hand, if the system is compiled using FreeRTOS, “fsl_i2c_freertos.h” is used. These files should be provided by the manufacturer.

```
1    #include "i2c_a7.h"
2    #include "fsl_clock.h"
3    #include "fsl_i2c.h"
4    #if defined(SDK_OS_FREE_RTOS) && SDK_OS_FREE_RTOS == 1
5    #include "fsl_i2c_freertos.h"
6    #endif
7    #include "fsl_port.h"
8    #include "sm_timer.h"
9    #include <stdio.h>
10   #include "fsl_gpio.h"
11   #include "sci2c_cfg.h"
```

**Fig 12. Necessary includes for RTOS or bare metal systems**

## 6.3 Adapting timers

The timers are defined by the system and are used to manage interruptions. Fig 13 illustrates the example in this guide, using as target platform Linux, RTOS or bare metal. It is more straightforward to implement “sm_timer.c” in Linux or RTOS versions as those timers are usually implemented natively. In a bare metal system, the timers are defined by the MCU manufacturer.

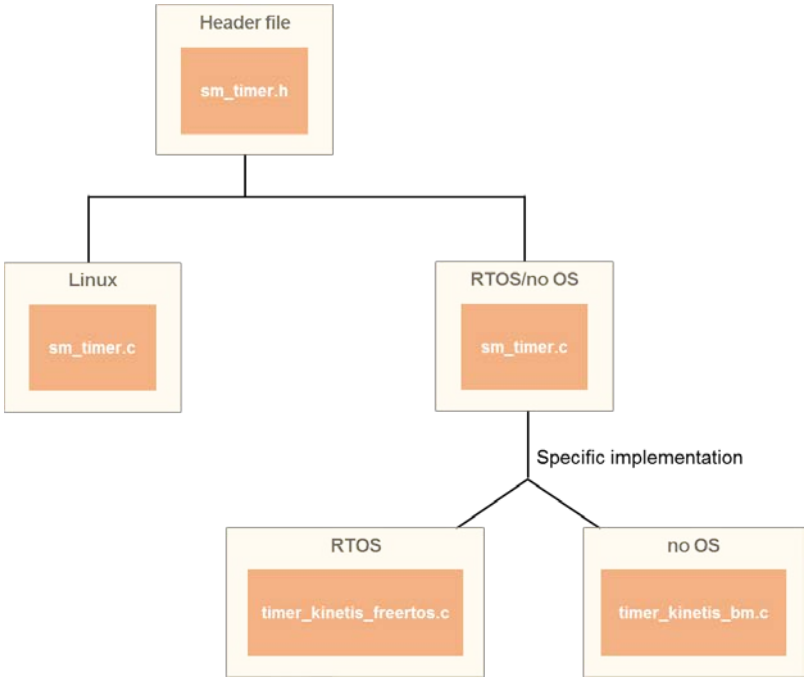


**Fig 13. Sm_timer.c implementation in each target platform**

The A71CH Host Library provides the ‘sm_timer.c’ file already adapted for the Kinetis family. Fig 14 shows the implementation of the ‘sm_usleep()’ function for this case. As can be observed, the sleep time is defined by the macro CORRECTION_TOLERENCE. This macro depends on the Complier used and the core clock frequency of the Kinetis MCU used (FRDM_KW41Z, FRDM_K82F or FRDM_K64F).

```
1    void sm_usleep(uint32_t microsec) {
2        gusleep_delay = microsec * CORRECTION_TOLERENCE;
3        while (gusleep_delay-- ) {
4            __NOP();
5        }
6    }
```

**Fig 14. Sm_timer.c implementation for the Kinetis family.**

It is important to remark that the Host MCU must respect the delays specified in the SCI2C protocol [SCI2C]. Current code uses microsecond level delays using software loop (highlighted in red). Depending on the clock and complier, a very fast Host would

violate the protocol and result in a non-responsive system. It is recommended to use Hardware timers if available. However, a hardware clock may lead to resource conflicts for system integration and, hence, they are not used in this implementation.

In addition, there are two possible sub-scenarios:

- **RTOS**: RTOS has its own timers defined; in this example, a Kinetis MCU (Kinetis K64F) board with FreeRTOS is used. FreeRTOS libraries are included in sm_timer.c because, as with Linux, FreeRTOS allows waiting for a certain time. This time is defined by a number of 'systicks' (system ticks). In current integration, one 'systick' is configured to last 1 millisecond. Fig 15 illustrates the implementation in a Kinetis MCU. The FreeRTOS function "vTaskDelay" is employed to handle the interruption.

```
7    /* initializes the system tick counter
8     * return 0 on succes, 1 on failure */
9    uint32_t sm_initSleep() {
10       return 0;
11   }
12
13   /**
14    * Implement a blocking (for the calling thread) wait for a number of
     milliseconds.
15    */
16   void sm_sleep(uint32_t msec) {
17       vTaskDelay(msec);
18   }
19
20   void vApplicationTickHook() {
21       gtimer_kinetis_msticks++;
22   }
```

**Fig 15. Sm_timer.c implementation in a Kinetis board with FreeRTOS**

- **Bare metal**: A bare metal system is not controlled by any operating system (no OS). Timers should be implemented according to the specifications of the MCU; e.g., manually implemented. Fig 16 depicts the implementation of the function "sm_sleep" in a Kinetis MCU (K64F) without OS. The function highlighted in red is implemented according to the board specifications of the target platform to implement the sleep functionality. In bare metal systems, the systick timer is used to be triggered every 1 millisecond, and that is how the system keeps track of the delay.

```
1    static void systick_delay(const uint32_t delayTicks) {
2        uint32_t currentTicks;
3        assert(delayTicks < 0x7FFFFFFFu);
4
5        __disable_irq();
6
7        if ((gtimer_kinetis_msticks) & 0x80000000u)
8        {
9            /* gtimer_kinetis_msticks has increased drastically (MSB is set),
10            * So, reset gtimer_kinetis_msticks before it's too late to detect an
```

```
11            * overflow. */
12           gtimer_kinetis_msticks = 0;
13       }
14
15       currentTicks = gtimer_kinetis_msticks;   // read current tick counter
16
17       __DSB();
18       __enable_irq();
19
20       // Now loop until required number of ticks passes
21       while ((gtimer_kinetis_msticks - currentTicks) <= delayTicks) {
22   #ifdef __WFI
23           __WFI();
24   #endif
25       }
26   }
27
28   /* interrupt handler for system ticks */
29   void SysTick_Handler(void) {
30       gtimer_kinetis_msticks++;
31   }
32
33
34   /* initializes the system tick counter
35    * return 0 on succes, 1 on failure */
36   uint32_t sm_initSleep() {
37       gtimer_kinetis_msticks = 0;
38       SysTick_Config(SystemCoreClock / 1000);
39       __enable_irq();
40       return 0;
41   }
42
43   /**
44    * Implement a blocking (for the calling thread) wait for a number of
     milliseconds.
45    */
46   void sm_sleep(uint32_t msec) {
47       systick_delay(msec);
48   }
```

**Fig 16. Sm_timer.c implementation in a Kinetis MCU without OS**

**Note:** Starting in release 1.5 of the A71CH Host Library, support for the LPC54018 Family will be added. The ‘i2c_kinetis.c’ file will be named ‘i2c_frdm.c’ and it will be used for the FRDM boards, while the support for the LPC54018 family will be in the files named ‘i2c_lpc54xxx.c’.

# 7. Printing layer (sm_printf.h)

The last file to be implemented is “sm_printf.h”. It is used to print messages in a console in a debug mode, for example. The printing layer is usually platform independent, as it only needs two standard libraries. Fig 17 illustrates the implementation of “sm_printf” platform independent.

```
1    #include <stdio.h>
2    #include <stdarg.h>
3
4    #include "sm_printf.h"
5
6    #ifdef FREEDOM
7    #   include "fsl_device_registers.h"
8    #   include "fsl_debug_console.h"
9    #   include "board.h"
10   #else
11   #   define PRINTF printf
12   #endif
13
14   #define MAX_SER_BUF_SIZE   (1024)
15
16   void sm_printf(uint8_t dev, const char * format, ...)
17   {
18       uint8_t  buffer[MAX_SER_BUF_SIZE + 1];
19       va_list   vArgs;
20
21       dev = dev; // avoids warning; dev can be used to determine output channel
22
23       va_start(vArgs, format);
24   #ifdef _WIN32
25       vsnprintf_s((char *)buffer, MAX_SER_BUF_SIZE, MAX_SER_BUF_SIZE, (char
     const *)format, vArgs);
26   #else
27       vsnprintf((char *)buffer, MAX_SER_BUF_SIZE, (char const *)format, vArgs);
28   #endif
29       va_end(vArgs);
30
31       PRINTF("%s", buffer);
32   }
```

**Fig 17. Sm_printf.c implementation platform independent**

The “stdarg.h” library provides the functions needed to use a varArg list. Note that Windows has its own implementation of “vsnprintf_s”. The “stdio.h” library provides the “printf” function needed to print output data in a console. If the target platform doesn’t support the “stdio.h” or “starg.h”, the developer should implement the “sm_printf” function according to the target platform specifications.

# 8. References

All the references contained in this document are listed in the following table:

**Table 1. References**

| | |
|---|---|
| [SCI2C] | SCI2C Protocol Specification Rev 1.5 or later (restricted to 1.x Revision, NOT compatible with Rev. 2.x) |
| [SMBUS] | SMBus Protocol Specification, Available in official website - http://smbus.org/specs/ |
| [AN_SMBUS] | **AN4471 SMBus Quick Start Guide**, Application note, available in NXP website: https://www.nxp.com/docs/en/application-note/AN4471.pdf |
| [OPEN_SSL] | OpenSSL Cryptography and SSL/TLS Toolkit information. Available in the official website: www.openssl.org |
| [A71CH_DOXY] | Comprehensive – hyperlinked – documentation contained in the Host SW package. It covers both the Host Library API and the usage and application of the library. |
| [A71CH_APDU] | APDU Specification of A71CH Security Module (2.0) - ds409420 |
| [sw415612] | A71CH Solution Host SW Package. |
| [A71CH_SW_PACKAGE] | A71CH Host SW Package (1.1) - sw415612 |
| [A71CH_WORKING_SW] | AN - A71CH working SW application documentation |
| [AN_A71CH_HOST_SW] | **AN12133 A71CH Host Software Package Documentation**, Application Note, available in NXP website: https://www.nxp.com/docs/en/application-note/AN12133.pdf |

# 9. Legal information

## 9.1 Definitions

**Draft —** The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

## 9.2 Disclaimers

**Limited warranty and liability —** Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

**Right to make changes —** NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

**Suitability for use —** NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

**Applications —** Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

**Export control —** This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

**Translations —** A non-English (translated) version of a document is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

**Evaluation products —** This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

## 9.1 Licenses

**ICs with DPA Countermeasures functionality**

NXP ICs containing functionality implementing countermeasures to Differential Power Analysis and Simple Power Analysis are produced and sold under applicable license from Cryptography Research, Inc.

## 9.2 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

**FabKey —** is a trademark of NXP B.V.

**I²C-bus —** logo is a trademark of NXP B.V.

# 10. List of figures

## 11. List of tables

# 12. Contents





For more information, visit: http://www.nxp.com

**Date of release: 09 July 2018** **492410**

**Document identifier: AN12185**