Pro Active

SDK K531/INS

Developer's guide

PMDE100 rev. AA

CONTENTS

1. INTRODUCTION

1.1. PRODUCT BRIEF

Pro-Active K531 is an ISO/IEC 14443 coupler. As an OEM device, it provides an easy-to-use versatile interface between a computer or a microcontroller, and contactless cards or RFID tags.

Sometimes, it appears that embedding specific functions inside the K531 itself can be a great feature for the integrator : it can totally remove the need for an external host microcontroller, or at least allow to use a cheaper one –slower, smaller–, and it helps achieving the fastest transaction speed by dramatically reducing the number of exchanges between reader and host.

The SDK K531/INS is a set of source code and sample projects that make it easy to develop virtually any contactless-related application for the K531.

1.2. ABOUT THIS MANUAL

This manual is the reference guide for developers working on the K531 and its derivatives.

This document refers to release 2R of the coupler (K531-2R).
Earlier releases are not compliant with the K531/INS SDK.

Some parts of this manual focus on operating the device together with the cards that are supplied in the development kit :

- NXP Mifare Standard
- NXP Mifare UltraLite
- NXP Desfire
- INSEAL Jaycos.

Of course K531 is able to communicate with other kind of cards.

1.3. AUDIENCE

This developer's guide is designed for use by application developers. It assumes that the reader has expert knowledge of electronics and embedded software development, using the C language.

As the K531 module can virtually dialog with any ISO/IEC 14443 contactless card, it is better to have a good understanding of this standard[1].

More than that, working with a specific contactless card involves a complete understanding of the card itself. Please read carefully the datasheet and operating manual of the card(s) you plan to work with, to know clearly

- How the card must be operated at the contactless communication level (full or partial 14443 compliance, type A or type B)
- How the card must be operated at the application level (proprietary command set, ISO/IEC 7816-4 compliance, ...).

1.4. SUPPORT AND UPDATES

Interesting related materials (datasheet, application notes, sample softwares...) are available at Pro-Active's web site : www.pro-active.fr .

Updated versions of this document and others will be posted on this web site as soon as they are made available.

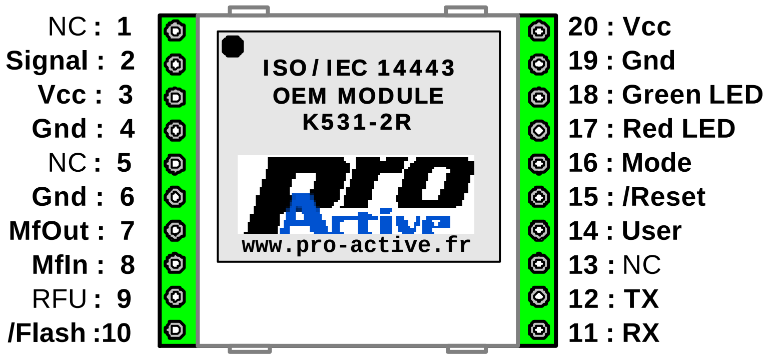
For technical support enquiries, please refer to Pro-Active support page, on the web at address www.pro-active.fr/support .

[1] International standard must be bought from ISO. Free drafts can be found at www.14443.org .

2. THE K531 PRODUCT FAMILY

2.1. K531 PINOUT AND BASIC OPERATION

a. *Pinout*



Module is powered by Vcc = 5V.

Connect antenna between pin “Signal” (2) and Gnd. Follow datasheets and application notes for details.

Pin “/Reset” (15) must be set to Vcc (or left unconnected) for operation.

Pin “/Flash” (10) must be set to Vcc (or left unconnected) for standard operation. Set “/Flash” to Gnd only when you want to download a new firmware into reader’s flash memory (ROM).

b. *Serial communication*

Serial communications uses UART 1 of the MCU. Pin “TX” (12) is the output (MCU to host) and pin “RX” (11) the input (host to MCU).

Both pins are 0-5V. An external line driver is required for RS-232 operation (or RS-422 or RS-485).

c. *I/Os*

- Pin 17 is an output only. Default implementation is to connect it to a green LED.
- Pin 18 is an output only. Default implementation is to connect it to a red LED.

Don’t connect pin 17 & 18 directly to the LEDs, a LED driver is required (see relevant application note).

- “User” (14) is either input or and output depending on software.
- “Mode” (16) is either input or and output depending on software.

When using pin 14 or pin 16 as output, observe same precautions as for pins 17 & 18.

d. MfIn / MfOut

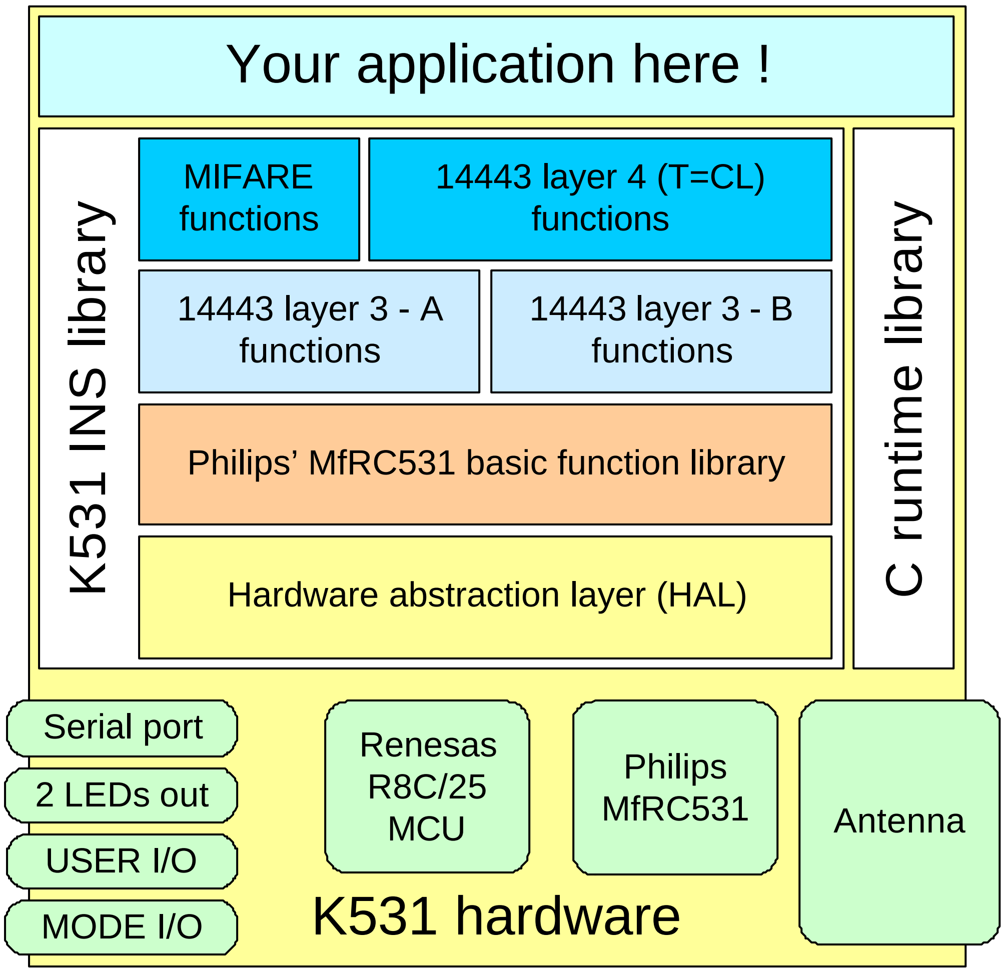
Those pins are directly connected to NXP RC531' MfIn & MfOut pins. Refer to NXP documentation for information.

2.2. PROGRAMMING MODEL

The diagram below depicts the K531 programming model.

You've complete control on the actual application, starting at function **`main`** entrance (and typically never exiting !).

The underlying complexity of the NXP MfRC531 chipset and of the ISO/IEC 14443 standard is totally hidden by the **K531 INS library**. The hardware abstraction layer and the standard C runtime library let you focus on the core of your project.



2.2.1. About the K531 INS library

The **K531 INS library** is a complete set of function that gives you complete and easy access to all the feature of the K531 and its ISO/IEC 14443 contactless interface.

> Thanks on an adaptive hardware abstraction layer, all Pro-Active's contactless products are build on the same core library, from inexpensive K531 OEM modules, to mobile SpringProx couplers for PocketPC and desktop CSB products.
>
> The **K531 INS library** is the K531-2R build of this core library.
>
> Once your application has been written with this SDK, it can virtually be ported to any other Pro-Active device. Don't hesitate to contact us if you think you can embed your application in our others products.

The documentation of the library is located in the `docs/k531_ins` directory of this SDK, listing all function prototypes and available features.

👍 This document provides a few useful examples but doesn't cover all the functions.

Only the `docs/k531_ins` documentation is the reference for function prototypes, return values and potential side-effects.

2.2.2. Differences with Pro-Active out-of-shelf products

Pro-Active contactless products (K531, CSB, SpringProx…) are also built on top of the same library, as your project will.

The difference is lying in the « your application here ! » panel, where Pro-Active puts its host communication layers (modified OSI 3994, fast binary, ASCII) and its console processor.

Due to the limited size of memory available in K531, we can't embed both « your application » and our host communication layer, that make our devices work with our host-based SpringProx API.

2.3. VARIOUS HARDWARE

The K531 module can be used on different hardware configurations. The common part is the contactless antenna, which must be designed with care for proper operation.

2.3.1. RS232 or USB serial line

In this typical configuration, the RX/TX pins are bound to an RS232 line driver (MAX232 or alike), or to an USB ←→ serial bridge (FTDI FD232 or alike).

This is for instance the configuration provided by K531-232 board (OEM antenna with RS232 link), IWM-K531-232 (wall-mount reader with 232 link), IWM-K531-USB (wall-mount reader with USB link).

2.3.2. Antennas with RS485 link

In this typical configuration, the RX/TX pins are bound to an RS485 line driver. Transmit mode is driven by pin “Mode”.

This is for instance the configuration provided by K531-485 board (OEM antenna with RS485 link) and IWM-K531-485 (wall-mount reader with 485 link).

2.3.3. Antennas with Dataclock or Wiegand lines

In this typical configuration, the UART is disabled. RX/TX pins outputs only, and deliver either an ISO2 (data+clock) or a Wiegand (D0+D1) signal.

IWM-K531-485 motherboard can be configured to provide this feature.

3. THE RENESAS R8C/25 MCU

The core of the K531 is the Renesas R8C/25 MCU.

Please download the R8C/25 hardware manual from Renesas' web site :

- http://www.renesas.com → « Global site »,
- « M32C/M16C/R8C » → « M16C » → « R8C/Tiny » → « R8C/25 group ».
- Choose the « R8C/24, R8C/25 group hardware manual » in the documentation page.

This is the current URL of the manual :

http://documentation.renesas.com/eng/products/mpumcu/rej09b0244_r8c2425hm.pdf

Be careful that it may be moved to another URL at any time.

3.1. SYSTEM MEMORY

Reference	Renesas R8C/25 MCU	Program Flash ("ROM")	Data Flash	RAM
K531-2R	R5F21256	32kB	2kB	2kB

3.1.1. *How compiler maps each item*

- Program code goes into Program Flash (sections : program, switch table, interrupts),
- Constants (C "const" keyword) go into Program Flash (sections rom_xx) ,
- Un-initialised global or static variables go into RAM (sections bss_xx ; they are implicitly initialised at 0 on start-up),
- Initialised global or static variables go both into RAM –where they are used– and into Program Flash –where their initial value is stored– (sections data_xx) ,
- Automatic variables are allocated on the stack. Their initial value –if some– is embedded in the program itself.

The Data Flash provides a persistent storage ("FEED"), see chapter 8.

3.1.2. *Important note regarding stack*

The R8C/25 has two different stack pointers :

- The USER stack pointer is used by the application,
- The SYSTEM stack pointer is used by the interrupt handlers.

When your code calls a function from the **K531 INS library** or from the C-runtime library, it will use either the USER stack or the SYSTEM stack, depending on the context of the caller.

All the examples provided in this SDK are configured as follow :

- 768B of USER stack
- 128B of SYSTEM stack.

High-level **K531 INS library** functions need at least 512B of stack memory to accommodate nested calls. Never call a **K531 INS library** function from an interrupt handler since the SYSTEM stack is far too small.

Avoid recursive functions.

Forbid oversized automatic variables.

Use the `static` keyword to put local variables outside the stack whenever it is possible.

You can change the stack settings in HEW :

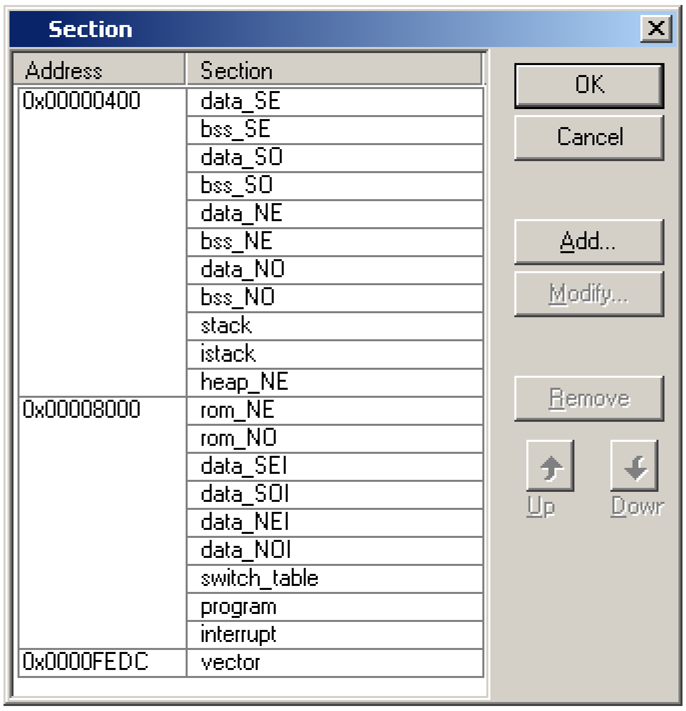
- « Build » menu,
- « Renesas M16C Standard Toolchain » menu item,
- « C » section,
- In the « Options C » text box, edit defines __STACKSIZE__ (USER stack) and __ISTACKSIZE__ (SYSTEM stack).

Setting __STACKSIZE__ to a value less than 728B (0x300) or __ISTACKSIZE__ to a value less than 128B (0x80) is not recommended.

3.1.3. *Accessing the memory mapping*

If you want to verify or modify the memory mapping :

- « Build » menu,
- « Renesas M16C Standard Toolchain » menu item,
- « Link » section,
- « Section Order » category.



Address	Section
0x00000400	data_SE
	bss_SE
	data_SO
	bss_SO
	data_NE
	bss_NE
	data_NO
	bss_NO
	stack
	istack
	heap_NE
0x00008000	rom_NE
	rom_NO
	data_SEI
	data_SOI
	data_NEI
	data_NOI
	switch_table
	program
	interrupt
0x0000FEDC	vector

3.1.4. *A few important hints*

- The memory mapping used in this SDK doesn't reserve memory for the heap (C dynamic allocation features).

Never call a function that performs dynamic allocation (`malloc`, `strdup`, ...).

- Don't waste RAM when ROM can be used.

Use the `const` keyword to put the « non variable variables » into ROM instead of RAM whenever it is possible.

- Don't waste RAM with two static or global variables that are never used in the same time.

Suppose function_A works on big_var_A, and function_B on big_var_B, where big_var_A and big_var_B are 1kB buffer (too big for the stack, of course).

If you can make sure that function_A never calls function_B, and function_B never calls function_A, use an union to store the two buffers at the same place in RAM.

- Some functions from the C runtime library are really huge in ROM.

String formatting functions (`sprintf` and alike) have a really high memory footprint. Whenever possible, try to rewrite the features you need cleverly, instead of calling such functions.

3.2. K531 IMPLEMENTATION SPECIFICS

3.2.1. *Reserved peripherals*

The K531 hardware gives strong limitations to the R8C/25 peripherals of the R8C/25 left available. Moreover, the **K531 INS library** uses most of the peripherals through its Hardware Abstraction Layer ; the application should always call the library functions instead of trying to gain direct access to the hardware.

Peripheral	Owner	Remarks
Timer RA	LIB	Use `timeout_` functions.
Timer RB		Never activate TRB0
Timer RD		Never activate TRDIO
Timer RE		Never activate TRE0
Port P0 A/D converter		
Port P1 UART 0	HAL	RC531 address bus
Port P2 Timer RD I/Os	HAL	RC531 data bus
Port P3 TRA0 & TRB0 I2C/SPI	HAL	RC531 control
Port P4 INT0 & INT1	HAL HAL	Clock input RC531 control
Port P6 UART 1 TRE0	HAL & LIB LIB	I/O pins & RC531 control Use `serial_1_` and `print_` functions

R8C/25 peripherals – Greyed items are not available to the application developer

3.2.2. *Clock frequency and main timer*

In K531 the R8C/25 runs exactly at 13,56MHz (same frequency as RF field for contactless communication).

Timer RA is reserved by the library and provides a base time of 1kHz (1ms period). The `timer_ticks` global variable (`DWORD`) is set to 0 on start-up, and increased by 1 every millisecond. It is available through `timout_init`, `timeout_expired` and `timeout_kill` functions or macros.

3.2.3. *Serial line*

UART 1 peripheral is bound to K531's "RX" and "TX" pins. The `serial_1_init` function configures the UART for 8 data bits, 1 stop bit, no parity, no flow control operation. The baudrate is specified in function call. Available baudrate are :

- 1200bps,

- 4800bps,
- 9600bps,
- 19200bps,
- 38400bps,
- 57600bps,
- 115200bps[2].

The **K531 INS library** doesn't allow any other baudrate.

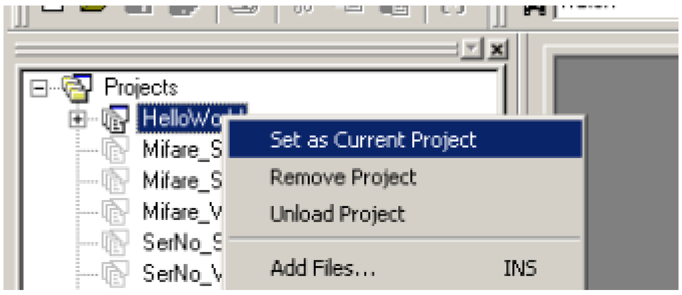
3.2.4. RC531 chipset

In K531, R8C/25 MCU communicates with NXP' MfRC531 contactless chipset through a high-speed parallel link. This allows fast communication in both directions. Anyway, when working with high-speed contactless cards (848kbps T=CL communication), the R8C/25 may be unable to empty or fill-in the MfRC531 FIFO buffer at desired speed. Please contact us if you need help on this subject.

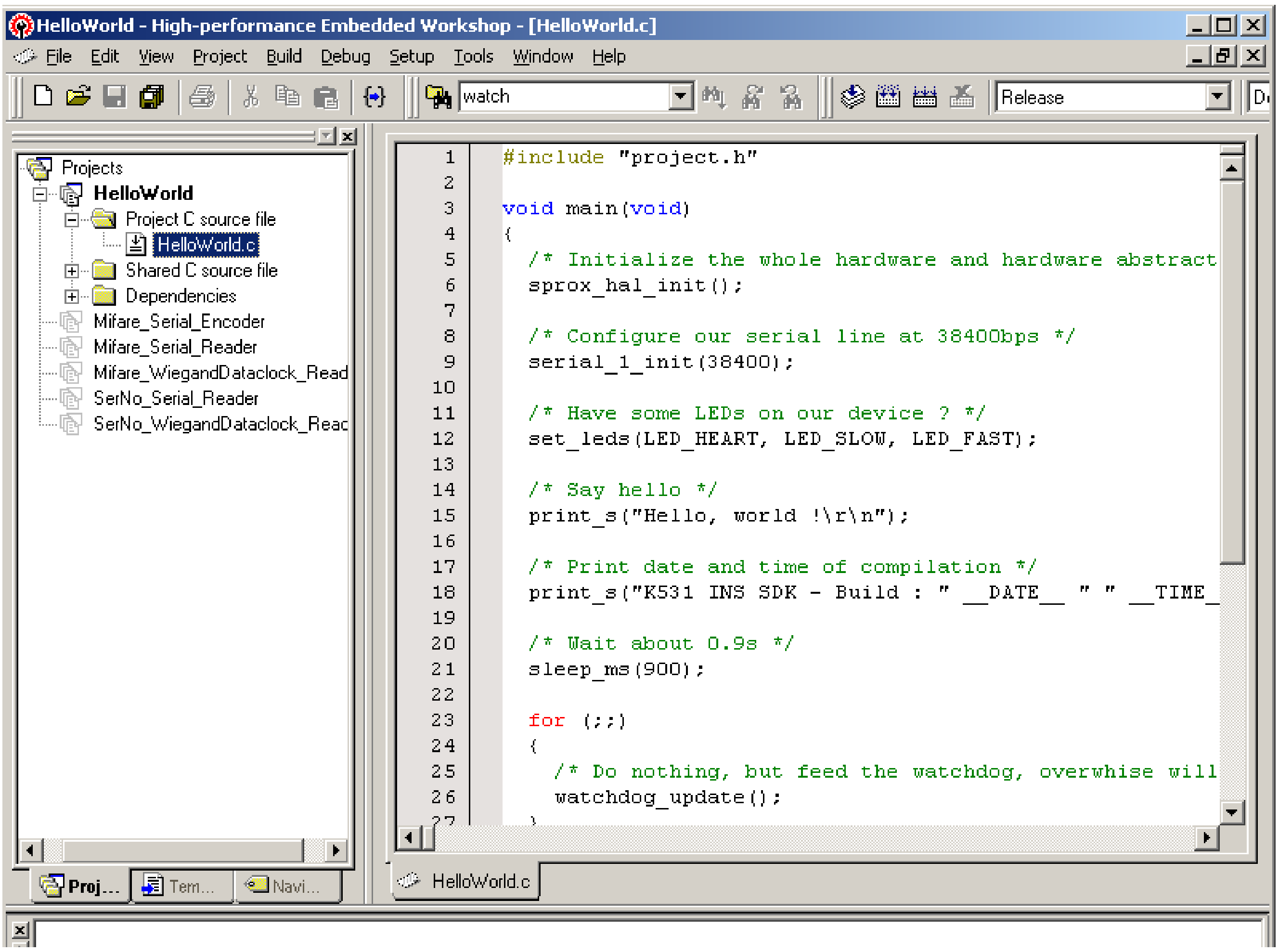
[2] To achieve this baudrate, an external crystal is required. It is available on K531 devices, but for example SpringProx PocketPC products are not equipped with it and will hang if the application tries to initialize their UART at this speed.

4. THE HELLOWORLD PROJECT

- Launch Renesas HEW 4, and open workspace
  C:\Renesas\pro-active_k531_ins_r8c-25\projects\Projects.hws
- Select the **HelloWorld** project in HEW projects explorer. Right-click, and click **Set as current project** in the popup menu to start working on this project.



- Explore project HelloWorld, and open the **HelloWorld.c** source file.



```
#include "project.h"

void main(void)
{
  /* Initialize the whole hardware and hardware abstract
  sprox_hal_init();

  /* Configure our serial line at 38400bps */
  serial_1_init(38400);

  /* Have some LEDs on our device ? */
  set_leds(LED_HEART, LED_SLOW, LED_FAST);

  /* Say hello */
  print_s("Hello, world !\r\n");

  /* Print date and time of compilation */
  print_s("K531 INS SDK - Build : " __DATE__ " " __TIME_

  /* Wait about 0.9s */
  sleep_ms(900);

  for (;;)
  {
    /* Do nothing, but feed the watchdog, overwhise will
    watchdog_update();
```

4.1. CODE REVIEW

This is the source code of the **HelloWorld's main**, without the comments :

```
void main(void)
{
  sprox_hal_init();
  serial_1_init(38400);
  set_leds(LED_HEART, LED_SLOW, LED_FAST);
  print_s("Hello, world !\r\n");
  print_s("K531 INS SDK - " __DATE__ " " __TIME__ "\r\n");
  for (;;)
    watchdog_update();
}
```

- At first we call `sprox_hal_init` to configure the Hardware Abstraction Layer.
- We initialise the serial line at 38400bps.
- The `print_s` function sends a string on the serial line.

The C runtime library provides standard `printf` function, but the stdin, stdout and stderr streams are not bound to the serial line. In other words, you can call `printf`, but it will do nothing at all (but waste a lot of ROM).

- The `for(;;)` statement is an infinite loop, our application will do nothing, and do it forever (at least, until we remove power or the device resets[3]).

On start-up, the **K531 INS library** activates the hardware watchdog of the MCU.

The overflow period of the watchdog is about 310ms on K531 (but can be as short as 150ms on other devices of the family).

Call the `watchdog_update` in main loop to confirm everything is OK, and at least every 50ms if you're ever caught in a function that may last longer.

[3] The K531 INS library systematically resets when the tick counter overflows. That is every 0xFFFFFFFF millisecond. If you develop a product that will be always ON –such as an access control reader- it will reset every 49,71 days. Since the typical reset sequence is shorter than 100ms, the reset will remain virtually undetectable by the end-user, until your application reports loudly.

Now consider the second functions in this file :

```
void serial_1_recv_callback(BYTE r)
{
  print_b(r);
}
```

This is the callback function that will be called by the **K531 INS library** each time a character is received on the serial line.

In this sample we simply call `print_b` to echo back the received character.

☠ Keep in mind that `serial_1_recv_callback` is an interrupt handler. This has 2 implications :

- SYSTEM stack is selected, ***never*** call any stack consuming function from this call-back, it will overflow,
- Until you've returned from this callback, the serial receive interrupt is inhibited. This means that if you spend too much time processing one byte, you'll lose the next byte (**overrun** error).

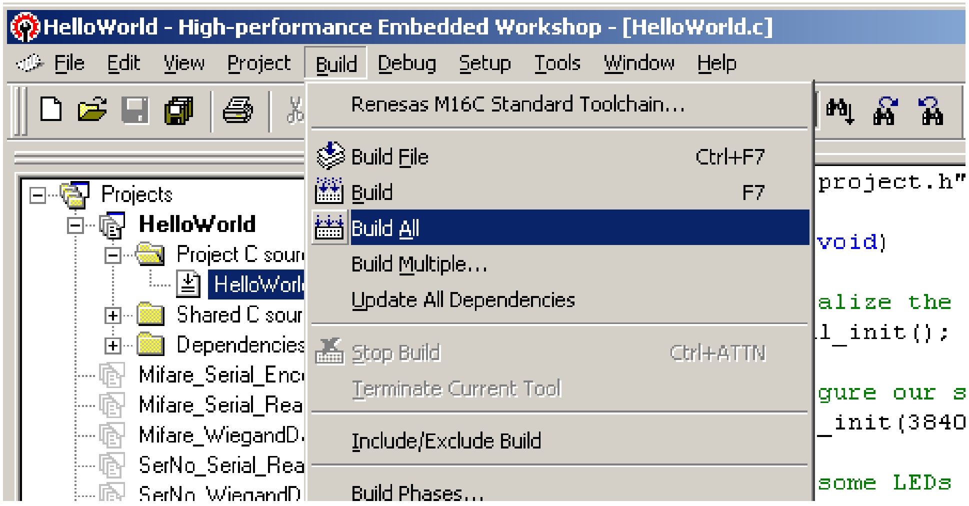
Chapter 9 gives an example of how-to implement a "console" on the serial line safely.

There's a second interrupt handler (`serial_1_error_callback`) that is called whenever a communication error is reported by the UART. The overrun flag tells you that the error is due to `serial_1_recv_callback` taking to much time.

On communication error, we reset the UART (call `serial_1_init` again).

4.2. BUILDING THE PROJECT

- In main menu, click **Build → Build All**.



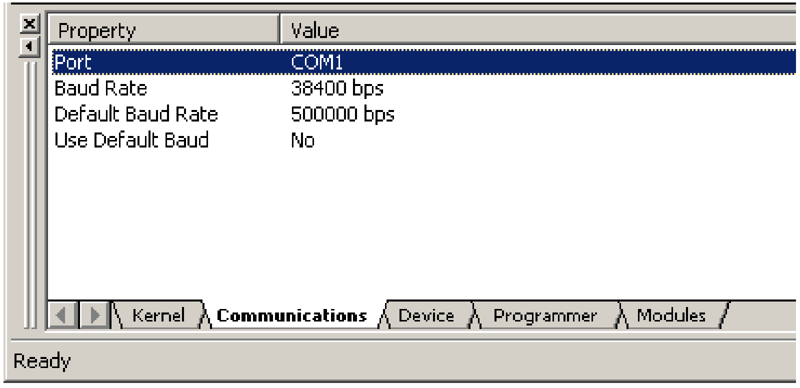
Here’s the build output :

```
Building All - HelloWorld - Release

Phase M16C C Compiler starting
C:\Renesas\Pro-Active_K531_INS_R8C-25\Projects\HelloWorld\HelloWorld.c
C:\Renesas\Pro-Active_K531_INS_R8C-25\Projects\HelloWorld\HelloWorld.c
C:\Renesas\Pro-Active_K531_INS_R8C-25\Sources\vectors.c
C:\Renesas\Pro-Active_K531_INS_R8C-25\Sources\vectors.c
C:\Renesas\Pro-Active_K531_INS_R8C-25\Sources\lowinit.c
C:\Renesas\Pro-Active_K531_INS_R8C-25\Sources\lowinit.c
C:\Renesas\Pro-Active_K531_INS_R8C-25\Sources\corks.c
C:\Renesas\Pro-Active_K531_INS_R8C-25\Sources\corks.c
Phase M16C C Compiler finished

Phase M16C Linker starting
Linkage Editor (ln30) for R8C/Tiny,M16C Series Version 5.12.02.000
Copyright(C) 2005. Renesas Technology Corp.
and Renesas Solutions Corp., All Rights Reserved.
now processing pass 1
processing "C:\Renesas\Pro-Active_K531_INS_R8C-25\Projects\HelloWorld\Release\corks.r30"
processing "C:\Renesas\Pro-Active_K531_INS_R8C-25\Projects\HelloWorld\Release\HelloWorld.r30"
processing "C:\Renesas\Pro-Active_K531_INS_R8C-25\Projects\HelloWorld\Release\lowinit.r30"
processing "C:\Renesas\Pro-Active_K531_INS_R8C-25\Projects\HelloWorld\Release\vectors.r30"
processing "Libraries"
processing "Libraries"
now processing pass 2
processing "C:\Renesas\Pro-Active_K531_INS_R8C-25\Projects\HelloWorld\Release\corks.r30"
processing "C:\Renesas\Pro-Active_K531_INS_R8C-25\Projects\HelloWorld\Release\HelloWorld.r30"
processing "C:\Renesas\Pro-Active_K531_INS_R8C-25\Projects\HelloWorld\Release\lowinit.r30"
processing "C:\Renesas\Pro-Active_K531_INS_R8C-25\Projects\HelloWorld\Release\vectors.r30"
processing "C:\Renesas\pro-active_k531_ins_r8c-25\projects\..\Library\r8c-25_k531_ins.lib ( sprox_ios_ins.r30 )"
processing "C:\Renesas\pro-active_k531_ins_r8c-25\projects\..\Library\r8c-25_k531_ins.lib ( sprox_con_ins.r30 )"
processing "C:\Renesas\pro-active_k531_ins_r8c-25\projects\..\Library\r8c-25_k531_ins.lib ( sprox_tmr_ins.r30 )"
processing "C:\Renesas\pro-active_k531_ins_r8c-25\projects\..\Library\r8c-25_k531_ins.lib ( sprox_hal_r8c-25_ins.r30 )"
processing "C:\Renesas\pro-active_k531_ins_r8c-25\projects\..\Library\r8c-25_k531_ins.lib ( rc500_drv.r30 )"
processing "C:\Renesas\pro-active_k531_ins_r8c-25\projects\..\Library\r8c-25_k531_ins.lib ( rc500_cfg.r30 )"
processing "C:\Renesas\pro-active_k531_ins_r8c-25\projects\..\Library\r8c-25_k531_ins.lib ( rc500_pcd.r30 )"
processing "C:\Renesas\pro-active_k531_ins_r8c-25\projects\..\Library\r8c-25_k531_ins.lib ( rc500_mio.r30 )"
processing "C:\Renesas\pro-active_k531_ins_r8c-25\projects\..\Library\r8c-25_k531_ins.lib ( serial_1.r30 )"
processing "C:\Renesas\pro-active_k531_ins_r8c-25\projects\..\Library\r8c-25_k531_ins.lib ( timer_ra.r30 )"
processing "C:\Renesas\pro-active_k531_ins_r8c-25\projects\..\Library\r8c-25_k531_ins.lib ( watchdog.r30 )"
processing "C:\Renesas\NC30WA\V540R00\lib30\r8clib.lib ( _i4divu.r30 )"
processing "C:\Renesas\NC30WA\V540R00\lib30\r8clib.lib ( _i4modu.r30 )"
processing "C:\Renesas\NC30WA\V540R00\lib30\r8clib.lib ( _i4mulu.r30 )"
processing "C:\Renesas\NC30WA\V540R00\lib30\r8clib.lib ( nmemset.r30 )"
Warning (ln30): License has expired, code limited to 64K (10000H) Byte(s)
DATA     0001266(004F2H) Byte(s)
ROMDATA  0000464(001D0H) Byte(s)
CODE     0006489(01959H) Byte(s)
The value of option function select register is FFH
Phase M16C Linker finished

Phase M16C Load Module Converter starting
Load Module Converter (lmc30) for R8C/Tiny,M16C/60 Series Version
4.01.01.000
Copyright(C) 2005. Renesas Technology Corp.
and Renesas Solutions Corp., All Rights Reserved.
--
Phase M16C Load Module Converter finished

Build Finished
0 Errors, 1 Warning
```

- Verify that build has finished with 0 error, and 0 or 1 warning[4].

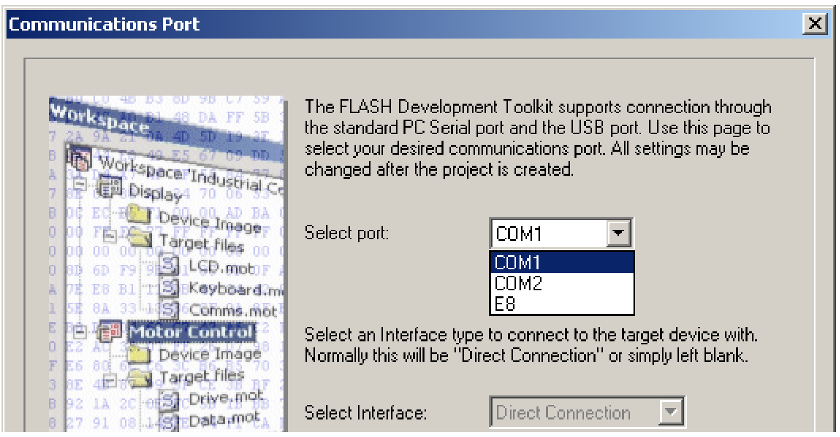
4.3. FLASHING THE DEVICE

- Launch Renesas FDT 3, and open workspace
  C:\Renesas\pro-active_k531_ins_r8c-25\flash\Projects.aws
- In main menu, click **Device → Configure flash project**.
- Go to the **Communications** tab in project's configuration.



Property	Value
Port	COM1
Baud Rate	38400 bps
Default Baud Rate	500000 bps
Use Default Baud	No

- Check that the selected serial port is the one your device is connected to. If not, double-click the **Port** line, and select appropriate port.



Communications Port

The FLASH Development Toolkit supports connection through the standard PC Serial port and the USB port. Use this page to select your desired communications port. All settings may be changed after the project is created.

Select port: COM1

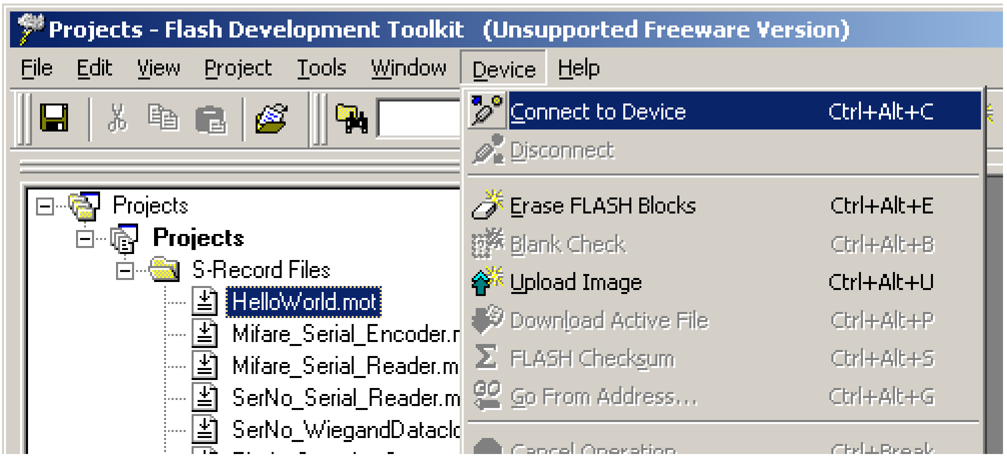
COM1
COM2
E8

Select an Interface type to connect to the target device with. Normally this will be "Direct Connection" or simply left blank.

Select Interface: Direct Connection

- Put your device in flash mode (refer to product manual for details).
- Select **HelloWorld.mot** in Projects → S-Record Files.
- In main menu, click, **Device → Connect to Device**.

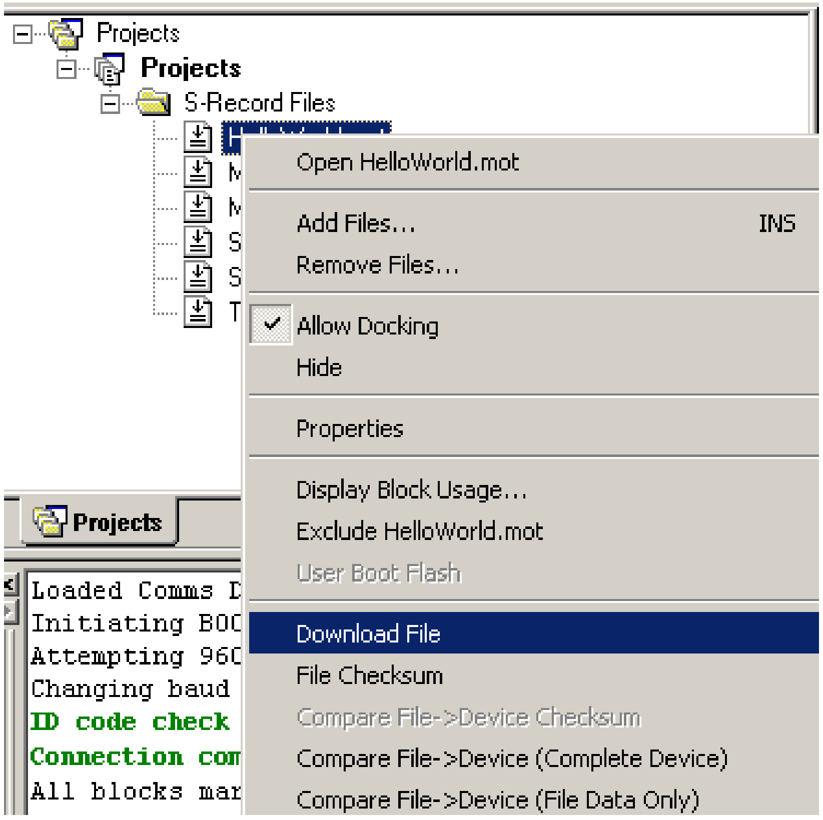
[4] The warning comes after 30 days when code size limit is enabled. This is not an issue since we work with a 32k MCU.


Projects - Flash Development Toolkit (Unsupported Freeware Version)
File Edit View Project Tools Window Device Help
Connect to Device Ctrl+Alt+C
Disconnect
Erase FLASH Blocks Ctrl+Alt+E
Blank Check Ctrl+Alt+B
Upload Image Ctrl+Alt+U
Download Active File Ctrl+Alt+P
FLASH Checksum Ctrl+Alt+S
Go From Address... Ctrl+Alt+G
Projects
Projects
S-Record Files
HelloWorld.mot
Mifare_Serial_Encoder.r
Mifare_Serial_Reader.m
SerNo_Serial_Reader.m
SerNo_WiegandDataclo

- Verify than connection is successful :

```
Connecting to device 'WS_R5F21256' on 'COM2'
Configuration:
'BOOT Mode' connection - using emulated interface
Opening port 'COM2' ...
Loading Comms DLL
Loaded Comms DLL
Initiating BOOT SCI sequence
Attempting 9600
Changing baud rate to 38400 bps
ID code check successful
Connection complete
All blocks marked as unknown written status
```

- Right-click HelloWorld.mot again, and click **Download file** in the popup.

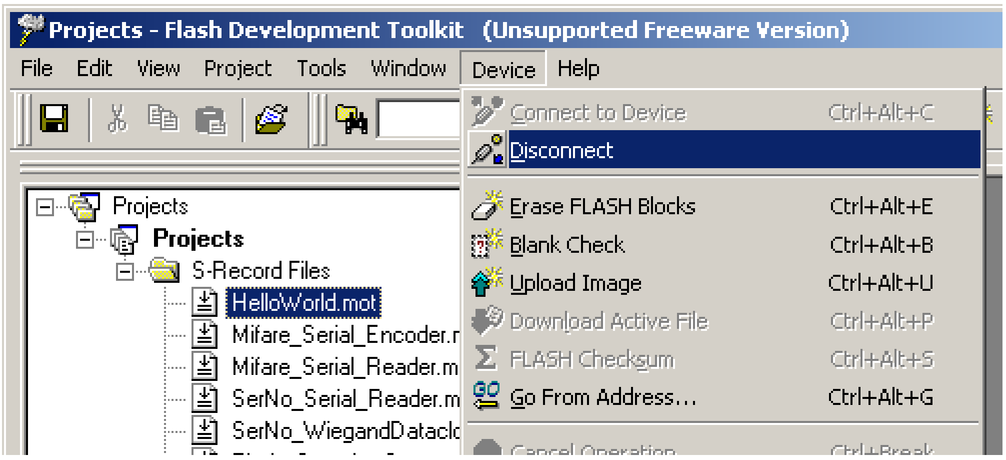


```
Loaded Comms D
Initiating BOO
Attempting 960
Changing baud
ID code check
Connection com
All blocks mar
```

- Verify that download is successful :

```
Erasing 2 blocks from device
Erased block EB1 (0x00008000 - 0x0000BFFF)
Erased block EB0 (0x0000C000 - 0x0000FFFF)
Erase complete

Processing file :"C:\Renesas\Pro-Active_K531_INS_R8C-25\Output\HelloWorld.mot"
[Data Flash] - No Data Loaded
Operation on User Flash
Writing image to device... [0x00008000 - 0x000099FF]
Writing image to device... [0x0000FE00 - 0x0000FFFF]
Data programmed at the following positions:
 0x00008000 - 0x000099FF      Length : 0x00001A00
 0x0000FE00 - 0x0000FFFF      Length : 0x00000200
7 K programmed in 3 seconds
Image successfully written to device
```

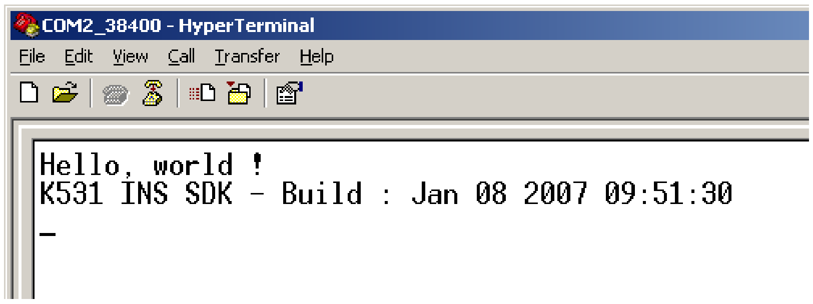
- In main menu, click, **Device** → **Disconnect**.


Projects - Flash Development Toolkit (Unsupported Freeware Version)
File Edit View Project Tools Window Device Help
Connect to Device Ctrl+Alt+C
Disconnect
Erase FLASH Blocks Ctrl+Alt+E
Blank Check Ctrl+Alt+B
Upload Image Ctrl+Alt+U
Download Active File Ctrl+Alt+P
FLASH Checksum Ctrl+Alt+S
Go From Address... Ctrl+Alt+G
Projects
Projects
S-Record Files
HelloWorld.mot
Mifare_Serial_Encoder.r
Mifare_Serial_Reader.m
SerNo_Serial_Reader.m
SerNo_WiegandDataclc

Renesas R8C-25 flash has a write endurance of 100 cycles.
This means that you can't reprogram your K531 more than 100 times.

4.4. TESTING OUR PROGRAM

- Launch HyperTerminal or any other terminal emulation software.
- Create a new connection to the serial communication port your device is connected to. Communication parameters are :
  - 38400 bps
  - 8 data bits, 1 stop bit
  - No parity, no flow control
- Put the device back in normal operation mode.
- Reset the device. The “Hello, world !” string will appear :



```
Hello, world !
K531 INS SDK - Build : Jan 08 2007 09:51:30
_
```

- Check that device echoes back the characters entered.

4.5. WHAT'S NEXT ?

You can modify this project to test various communication speeds. Note that any on communication error, the UART of the device is configured again ; don't forget to change baudrate in function `serial_1_error_callback` and not only in function `main`.

You can also try different LEDs commands, and also change LEDs behaviour dynamically when receiving specific characters.

Last but not least, see what happens when replacing the `watchdog_update()` statement in function `main`'s `for (;;)` loop by a `no_operation()` statement (calls NOP, i.e. does really nothing).

All the examples written in the next chapters are based of this HelloWorld example.

You can find each source code in the HelloWorld folder, with the name **`HelloWorld_<Chapter>_<Paragraph>.c`**

5. CONTACTLESS OPERATION

5.1. ACTIVATION OF THE RC531

Currently HelloWorld project only configure the MCU. First step is to attach and configure the NXP RC531 chipset, for actual contactless operation.

Just after `sprox_hal_init()`, call `rc531_connect()` to do so.

a. Updated code

Here's our `HelloWorld_5_1.c` . We've added a few lines to test RC531 :

```
void main(void)
{
  sprox_hal_init();
  rc531_connect();
  serial_1_init(38400);

  (...)

  /* Get and print RC531 info */
  /* ------------------------ */
  {
    BYTE buffer[5];
    SBYTE rc;

    print_s("RC531");
    rc = PcdGetPid(buffer); /* Retrieve product identifier */
    if (rc != MI_OK) print_d(rc, 0); /* Error */
    else
    {
      /* Display product identifier, 5 bytes */
      print_s(" PID="); print_h(buffer, 5, FALSE);
    }
    rc = PcdGetSnr(buffer); /* Retrieve serial number */
    if (rc != MI_OK) print_d(rc, 0); /* Error */
    else
    {
      /* Display serial number, 4 bytes */
      print_s(" SNR="); print_h(buffer, 4, FALSE);
    }
    print_s(NULL); /* Same as print_s("\r\n"); */
  }

  for (;;)
    (...)
```

b. Test program

Build updated program, disconnect terminal session, and flash the program. Connect again with the terminal emulator, and see new output :

```
RC531 PID=30FFFF0F04 SNR=17199747
```

5.2. LOOKING FOR A CARD

Now before working with a contactless card, we must find it the RF field. As the card doesn't say "Hello, I'm new here" when it arrives, reader must perform an active card detection, by sending repeated lookup frames. This continuous polling is the basis of a contactless reader.

5.2.1. *ISO/IEC 14443-A layer 3 activation*

There are two functions to lookup for an ISO/IEC 14443-A card :

- `IsoA_ActivateIdle` uses WUPA lookup frames, meaning that only "new" cards will answer.
- `IsoA_ActivateAny` uses REQA lookup frames, meaning that the cards that have previously been worked with and halted by the reader will answer again.

Both functions return `MI_OK` on success, and `MI_NOTAGERR` when no 14443-A card has been found in the RF field.

When result is `MI_OK`, the identification of the card is found in global variable `iso3a_tag`, which is an `ISO3A_TAG_ST` structure.

a. *Explanation of the `ISO3A_TAG_ST` structure.*

Field	Size (bytes)	Content
atq	2	Card's **Answer To Query**. This field provides information on the type of card we've found.[5]
uid	4, 7 or 12	Card's **Unique IDentifier** (UID). Size=4 for cards with a single-sized UID (Mifare 1k & 4k) Size=7 for cards with a double-sized UID (Mifare UltraLight & Desfire) Size=12 for cards with a triple-sized UID
uidlen	1	This is actual size of UID (4, 7 or 12)
sak	1	**Card's Select AKnowledge**. This field tells us whether the card supports 14443 layer 4 ("T=CL") operation or not.

[5] See NXP's application note "Mifare Interface Platform Type Identification Procedure" At the time of writing, this document can be found online at http://www.nxp.com/products/identification/mifare/index.html#rel

5.2.2. ISO/IEC 14443-B layer 4 activation

There are two functions to lookup for an ISO/IEC 14443-B card :

- `IsoB_ActivateIdle` uses WUPB lookup frames, meaning that only "new" cards will answer.
- `IsoB_ActivateAny` uses REQB lookup frames, meaning that the cards that have previously been worked with and halted by the reader will answer again.

Both functions take on parameter named `afi`.

Both functions return `MI_OK` on success, and `MI_NOTAGERR` when no 14443-B card has been found in the RF field.

When result is `MI_OK`, the identification of the card is found in global variable `iso3b_tag`, which is an `ISO3B_TAG_ST` structure.

a. Explanation of the `afi` parameter

Since 14443-B defines a really poor anti-collision scheme compared to 14443-A, when more than one card may be present in the RF field, it is easier to discover only the card we want to work with than trying to discover one after the other until we find the one we've been expecting.

The AFI (Application Family Identifier) represents the type of application targeted by the reader. Only cards (and hopefully only card_) with application(s) of the type indicated by the AFI are allowed to answer to REQB or WUPB. The list of AFIs a specific card will answer to, depends on the list of applications installed in the card.

If you want to lookup for any kind of 14443-B card, whatever the application they provide, set parameter `afi = 0`.

b. Explanation of the `ISO3B_TAG_ST` structure.

Field	Size (bytes)	Content
afi	1	Reminder of the AFI the card has answered to
atq	11	**Card's Answer To Query**. 4-first bytes of ATQ are named "Pseudo-Unique PICC Identifier" (PUPI). They can either be a 4-bytes fixed serial number, or a 4-byte random number changing on each activation. For explanation of the 7-next bytes, please refer to ISO/IEC 14443-3.

5.3. WORKED EXAMPLES

5.3.1. Basis

Here's our updated source code

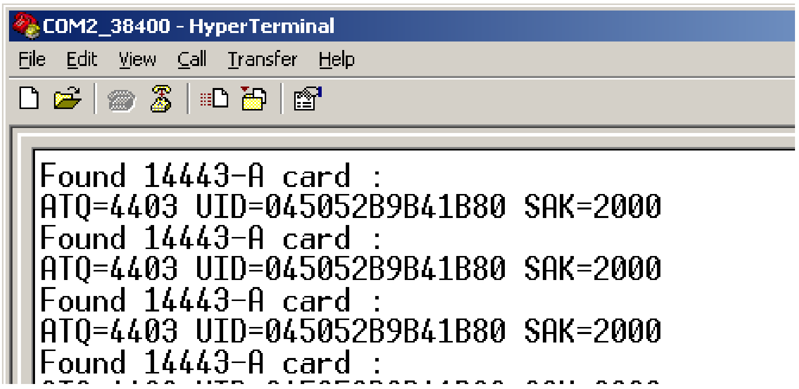
HelloWorld_5_3_1.c

```
    (...)

      for (;;)
      {
        SBYTE rc;

        /* Feed the watchdog */
        watchdog_update();

        /* 14443-A lookup */
        rc = IsoA_ActivateAny();
        if (rc == MI_OK)
        {
          print_s("Found 14443-A card :\r\n");
          print_s("ATQ=");
          print_h(iso3a_tag.atq, 2, FALSE);
          print_s(" UID=");
          print_h(iso3a_tag.uid, iso3a_tag.uidlen, FALSE);
          print_s(" SAK=");
          print_h(iso3a_tag.sak, 1, FALSE);
          print_s(NULL);
        }
        /* We must wait at least 5ms between each type */
        sleep_ms(5);

        /* 14443-B lookup, AFI = 0 (any application) */
        rc = IsoB_ActivateAny(0x00);
        if (rc == MI_OK)
        {
          print_s("Found 14443-B card :\r\n");
          print_s("ATQ=");
          print_h(iso3b_tag.atq, 11, FALSE);
          print_s(NULL);
        }
        /* We must wait at least 5ms between each type */
        sleep_ms(5);
      }
```

Once a card is found (either A or B), we display its information.

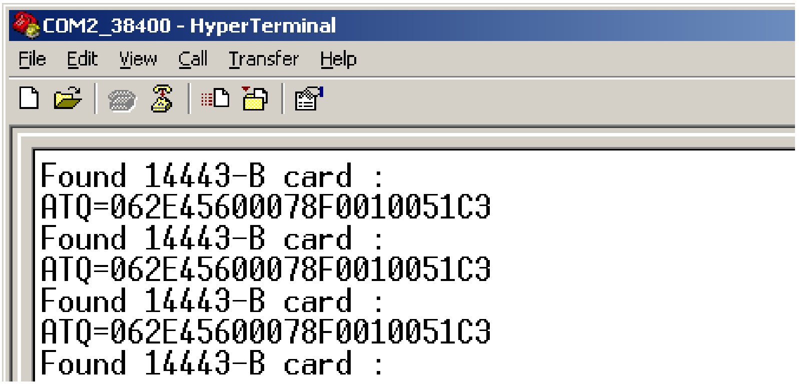
Note that we add a 5ms delays between type A and type B lookups. This is needed because ISO/IEC 14443 allows type B cards to reset after receiving a type A modulation, and type A cards to reset after receiving a type A modulation[6]. The standard allows 5ms for card being ready after a reset[7].

Here's the output when a type A card (NXP Desfire) is put in the field :



```
Found 14443-A card :
ATQ=4403 UID=045052B9B41B80 SAK=2000
Found 14443-A card :
ATQ=4403 UID=045052B9B41B80 SAK=2000
Found 14443-A card :
ATQ=4403 UID=045052B9B41B80 SAK=2000
Found 14443-A card :
```

Here's the output when a type B card (Inseal Jaycos) is put in the field :



```
Found 14443-B card :
ATQ=062E45600078F0010051C3
Found 14443-B card :
ATQ=062E45600078F0010051C3
Found 14443-B card :
ATQ=062E45600078F0010051C3
Found 14443-B card :
```

Observe that in both cases the information is repeated until card is removed from the field.

5.3.2. Type A anti-collision

Now we'll use the type A anti-collision feature, halting the card after having found it, and using REQA lookup instead of WUPA.

We'll try to do the same for type B, and check the differences.

[6] First case is really frequent, where the second has never been observed...

[7] Note that some "old" type B cards may require more than 5ms to wake-up after a field interruption. You'll have to adapt the timings of your reader to the requirements or the specific cards you're working with.

Here's the new source code for type A :

```
        /* 14443-A lookup, REQA instead of WUPA */
        rc = IsoA_ActivateIdle();
        if (rc == MI_OK)
        {
          IsoA_Halt(); /* Halt the card right now */
          print_s("Found 14443-A card :\r\n");

          (...)
```

Here's the new source code for type B :

```
        /* 14443-B lookup, REQB instead of WUPB */
        rc = IsoB_ActivateIdle(0x00);
        if (rc == MI_OK)
        {
          IsoB_Halt(iso3b_tag.atq); /* Halt the card right now */
          print_s("Found 14443-B card :\r\n");

          (...)
```

Complete code is in `HelloWorld_5_3_2.c`

Type B halt command is "addressed" to one specific card, so the PUPI (4-first bytes of ATQ) must be provided to `IsoB_Halt`.

Type A halt command is "broadcasted", but only the currently selected card will accept the command, that's why `IsoA_Halt` takes no parameter.

Now place a type A card in the field. The information is displayed once. Card must be removed and put back again to have its information displayed. You can also put 2 or 3 type A cards in the field in the same time, and see that type A anti-collision allows selecting one after the other.

Now place a type B card in the field. In most cases you'll see no difference with last version of the program, because the type B card resets (and forgets its "halted" state) during the type A modulation.

6. WORKING WITH MIFARE CARDS

This chapter deals with Mifare "Standard" (or Mifare "Classic") cards. The examples focus on Mifare 1k, but can be extended very easily to Mifare 4k[8].

6.1. RECOGNIZING MIFARE CARDS

Mifare cards can be recognized by their ATQ[9] :

iso3atag_atq[0]	iso3atag_atq[1]	Card
0x04	0x00	Mifare Standard 1k
0x02	0x00	Mifare Standard 4k

Here's a summary of the features :

a. *Mifare 1k*

- 64 blocks of 16 bytes each, blocks 0 is read-only.
- Card is divided into 16 sectors of 4 blocks each.
- Last block of each sector ("sector's trailer") stores the two secret keys (key A & key B) that protect this sector.

b. *Mifare 4k*

- 256 blocks of 16 bytes each, blocks 0 is read-only.
- Card is divided into 32 sectors of 4 blocks each (sectors 0 to 31), followed by 16 sectors of 16 blocks (sectors 32 to 39).
- Last block of each sector ("sector's trailer") stores the two secret keys (key A & key B) that protect this sector.

[8] The memory mapping under 2k is exactly the same, and after 2k only the number of blocs in a sector is different.

[9] See NXP's application note "Mifare Interface Platform Type Identification Procedure"
At the time of writing, this document can be found online at
http://www.nxp.com/products/identification/mifare/index.html#rel

This documents also specifies SAK = 0x08 for Mifare 1k and SAK = 0x18 for Mifare 4k. This is true for "real" NXP Mifare cards, but you can find a different SAK when using Mifare cards from other manufacturer (Infineon for instance) or when the card is a micro-controller smartcard, with a Mifare emulation applet (Mifare ProX cards for instance are often programmed with a Mifare Standard applet).

Note that this document –and a lot of documents written by Philips/NXP– considers ATQ as a single 16-bit value (a WORD) where the reader receives 2 8-bit values (2 BYTEs). The Mifare card is "little endian", so LSB maps to atq[0] and MSB to atq[1].

6.2. READING A BLOCK

K531 INS library provides 2 functions to read one block from a Mifare card :

- `MifReadWriteBlock(BOOL w, BYTE block, BYTE data[16], BYTE key_value[6])`
- `MifReadWriteBlockK(BOOL w, BYTE block, BYTE data[16], BYTE key_ident)`

In both functions the `w` parameter must be set to FALSE[10]. The `block` parameter is the address of the block to be read (0 to 63 for Mifare 1k, 0 to 255 for Mifare 4k) ; on success (function returning `MI_OK`) the `data` buffer will receive the actual data read from the card.

Thanks to Mifare security scheme, reading a block is only possible after a successful authentication, and communication is ciphered. Authentication is performed over the next security block (or sector's trailer) to be found after the specified `block`, and using a "secret" key.

Next paragraph provides details on Mifare keys.

Type B "halt" command (HLTB) is "addressed" to one specific card, so the PUPI (4-first bytes of ATQ) must be provided to `IsoB_Halt`.

Type A "halt" command (HLTA) is "broadcasted", but only the currently selected card will accept the command, that's why `IsoB_Halt` takes no parameter.

6.3. MIFARE ACCESS KEYS

Two keys protect each sector in a card :

- Key A is commonly used for read-only access,
- Key B is commonly used for read & write access.

Each key is a 6-byte value (48 bits)[11].

When reader wants to read one block, it must know either key A or key B of the sector this block belongs to.

[10] Set it to TRUE if you want to write the block instead of reading it.

[11] Although NXP documentation tells that Mifare is a "secure" contactless card, 48-bit keys are nowadays considered as really weak compared to 112 or 128-bit keys that are commonly used in 3-DES or AES operation. More than that, the CRYPTO1 security scheme used in Mifare authentication and secure communication is a proprietary algorithm, and nobody really knows how secure it really is. Anyway, in most "real-life" cases (access control, identification, …) where price of the solution is an important concern, the security level of Mifare cards & readers can be considered as really good compared to other solutions in the same range of prices (125kHz tags, 13.56MHz memory cards with no security at all…)

6.3.1. *Storing the keys in program*

The easiest solution is to store the key in program, and to provide it to `MifReadBlock` function :

```
static const BYTE my_key[6] = {0x12,0x34,0x56,0x78,0x9A,0xBC};

(...)

SBYTE rc = MifWriteReadBlock(FALSE, 4, data, my_key);
if (rc == MI_OK)
{
  /* Block 4 has been read !!! */
  print_h(data, 16, FALSE);

  (...)
```

When called with a non-NULL `key_value` parameter, function `MifReadWriteBlock` tries the specified key as a key A, and only on failure as a key B.

It is technically possible to read sector's trailer (blocks 3, 7, …), but this is not really interesting since access keys are "masked" by the card (read as 0x00 … whatever their value).

6.3.2. *Using RC531's secure EEPROM*

Thankfully, the RC531 chip has an internal secure non-volatile memory, where keys can be stored. The memory is said "secure" because one can write the keys in it, but never read them back (so the secret key is really secret).

Using the RC531 to store the key(s) has two interests :

- A stolen reader is no more a security concern ;
- The developer can test the application without any knowledge of the key. This is really interesting when the application is developed by a third-party ; actual Mifare key will be loaded only at deployment time, and will remains unknown from the third-party.

The counterpart is that one need to implement a mean of loading the keys into the RC531[12], either from serial line or through a configuration card[13].

[12] Having the embedded software loading the key at first boot is just kidding… Key is still to be found in source code and in binary dump…

[13] Well… Now you must find a way of securing the configuration cards, because they hold the key and because a forged configuration card will make your readers unusable…

a. *Without selection*

Once the key(s) are loaded into RC531's secure EEPROM, reading is possible without specifying the key :

```
(...)

/* Note the NULL key : */
SBYTE rc = MifWriteReadBlock(FALSE, 4, data, NULL);
if (rc == MI_OK)
{
  /* Block 4 has been read !!! */
  print_h(data, 16, FALSE);

  (...)
```

When called with a NULL `key_value` parameter, function `MifReadWriteBlock` will execute the following procedure :

- Tries sequentially all A keys from RC531's EEPROM, until one matches,
- If read all A keys have failed, tries sequentially all B keys, until one matches.

This is interesting for developer because it is easy to implement and because he doesn't need to know at design time which key index will actually be used, but leads to two issues :

- Since the RC531 has 16 A keys and 16 B keys, the complete procedure can take "a lot of time" before returning (namely 400ms if matching key is B 15).
- If read is successful, developer knows that the authentication has been successfully passed, but he doesn't know which key has been used. This is a potential security issue since a card may be read with a different key than the one specified.

b. *With forced key selection*

Using the `MifReadWriteBlockK` function, developer can specify which key index he wants to use for each particular block, thus knowing for sure the sector has been formatted with the expected key.

```
    (...)

    SBYTE rc = MifReadWriteBlockK(FALSE, 25, data,
                                  MIF_E2_KEY|MIF_KEY_A|0x07);
    if (rc == MI_OK)
    {
      /* Block 25 has been read with key A 7 from EEPROM */
      print_h(data, 16, FALSE);

      (...)
```

The `key_ident` parameter is a bit OR of :

- Constant MIF_E2_KEY to select RC531's EEPROM,
- Type of key is either MIF_KEY_A or MIF_KEY_B,
- Key index, from 0 to 15 (0x00 to 0x0F).

6.4. WORKED EXAMPLE

This example reads (and displays) content of block 25 with any of the EEPROM keys (no selection).

HelloWorld_6_4.c

a. *Pre-loading the keys*

We want to read the Mifare cards provided with the SDK. The card comes from manufacturer in a "transport" state that must be documented by the manufacturer. We assume they are in one of those two states :

- NXP configuration : key A is { 0xFF,0xFF,0xFF,0xFF,0xFF,0xFF } for every sector,
- Infineon configuration : key A is { 0xA0,0xA1,0xA2,0xA3,0xA4,0xA5 } for every sector.

So our program has to preload both keys to RC531's EEPROM (remember, this is not a good idea, actual keys must be loaded on-the-field over the serial line or through a configuration card). This is done once at the beginning.

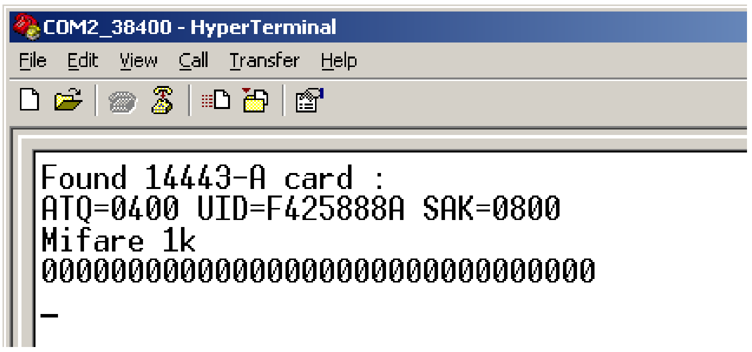
```
static const BYTE key_FF[6] = {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
static const BYTE key_Ax[6] = {0xA0,0xA1,0xA2,0xA3,0xA4,0xA5};

(...)

/* Store key_FF as key A, index 0 */
Mf500PcdLoadKeyE2(PICC_AUTHENT1A, 0, key_FF);

/* Store key_Ax as key A, index 1 */
Mf500PcdLoadKeyE2(PICC_AUTHENT1A, 1, key_Ax);
```

b. Updated 14443-A lookup code

```
rc = IsoA_ActivateIdle();
if (rc == MI_OK)
{
  print_s("Found 14443-A card :\r\n");
  print_s("ATQ=");
  print_h(iso3a_tag.atq, 2, FALSE);
  print_s(" UID=");
  print_h(iso3a_tag.uid, iso3a_tag.uidlen, FALSE);
  print_s(" SAK=");
  print_h(iso3a_tag.sak, 1, FALSE);
  print_s(NULL);

  /* Is this a Mifare card  ?*/
  if ( (iso3a_tag.atq[1] == 0x00)
   && ((iso3a_tag.atq[0] == 0x02)
   ||  (iso3a_tag.atq[0] == 0x04)) )
  {
    BYTE data[16];

    /* Yes ! */
    if (iso3a_tag.atq[0] == 0x04) print_s("Mifare 1k\r\n");
    if (iso3a_tag.atq[0] == 0x02) print_s("Mifare 4k\r\n");

    /* Read block 4 with any of the EEPROM keys */
    rc = MifReadWriteBlock(FALSE, 4, data, NULL);
    if (rc == MI_OK)
    {
      print_h(data, 16, FALSE);
      print_s(NULL);
    }
  }
  /* Halt the card only after Mifare processing */
  IsoA_Halt();
}
```

c. Output

Content of block 4 is displayed (out-of-factory cards come with all data set to 0x00, or sometimes with all data set to 0xFF) :



```
Found 14443-A card :
ATQ=0400 UID=F425888A SAK=0800
Mifare 1k
00000000000000000000000000000000
_
```

6.5. GOING FURTHER

6.5.1. *Writing data into the card*

Never write any sector's trailer (blocks 3, 7, ...) as you will overwrite sector's access keys and access conditions with your data. Setting invalid access conditions or forgetting the access keys permanently prevent any access to the sector !

In this example we read blocks 4 and 5, display both of them, and rewrite block 5 after altering its content. We need to preload the right B keys into RC531's EEPROM if we want the operation to succeed.

`HelloWorld_6_5_1.c`

a. Preloading the keys

```
static const BYTE key_Bx[6] = {0xB0,0xB1,0xB2,0xB3,0xB4,0xB5};

(...)

/* Store key_FF as key B, index 0 */
Mf500PcdLoadKeyE2(PICC_AUTHENT1B, 0, key_FF);

/* Store key_Bx as key B, index 1 */
Mf500PcdLoadKeyE2(PICC_AUTHENT1B, 1, key_Bx);
```

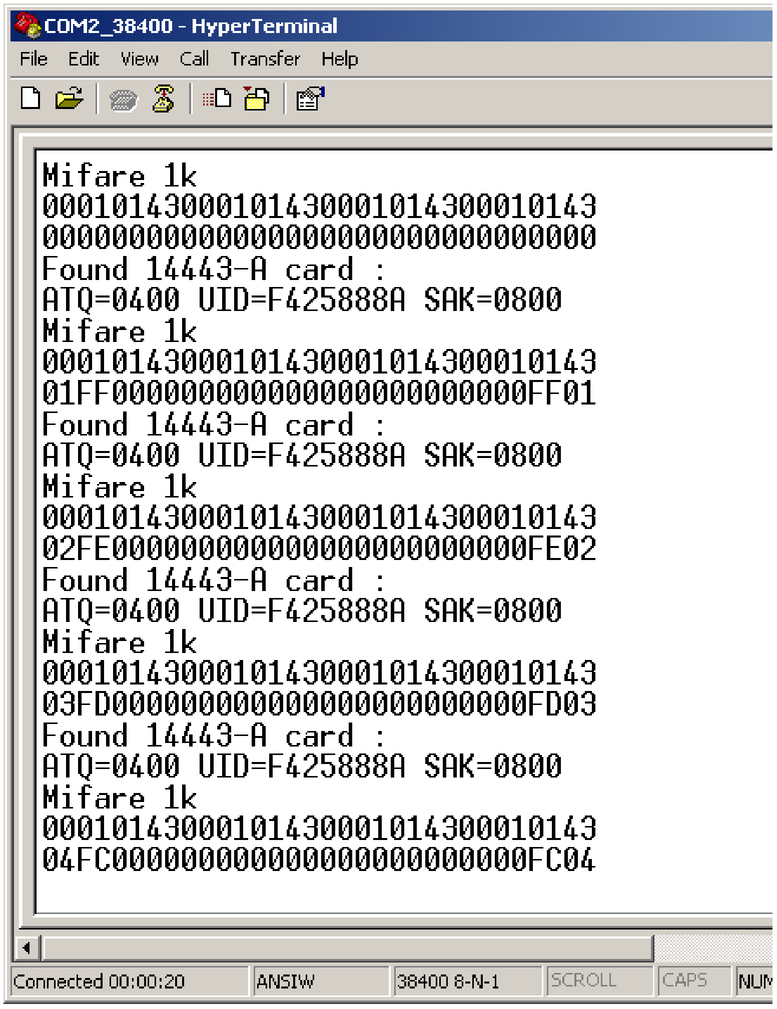
b. *Updated Mifare lookup code*

```
        rc = IsoA_ActivateIdle();
        if (rc == MI_OK)
        {
          (...)

          /* Is this a Mifare card  ?*/
          if ( (iso3a_tag.atq[1] == 0x00)
           && ((iso3a_tag.atq[0] == 0x02)
           ||  (iso3a_tag.atq[0] == 0x04)) )
          {
            (...)

            /* Read and display block 4 */
            rc = MifReadWriteBlock(FALSE, 4, data, NULL);
            (...)

            /* Read and display block 5 */
            rc = MifReadWriteBlock(FALSE, 4, data, NULL);
            (...)

            /* Do some changes in block 5 */
            data[0]++; data[15]++; data[1]--; data[14]--;

            /* Write back block 5 with any of the EEPROM keys */
            rc = MifReadWriteBlock(TRUE, 5, data, NULL);
            if (rc != MI_OK)
              print_s("Failed to rewrite block 5");
          }
          /* Halt the card only after Mifare processing */
          IsoA_Halt();
        }
```

c. *Output*

Content of block 5 is now displayed after content of block 4.

Note that content of block 5 is different every time we put the card on the antenna.



```
Mifare 1k
00010143000101430001014300010143
00000000000000000000000000000000
Found 14443-A card :
ATQ=0400 UID=F425888A SAK=0800
Mifare 1k
00010143000101430001014300010143
01FF00000000000000000000000FF01
Found 14443-A card :
ATQ=0400 UID=F425888A SAK=0800
Mifare 1k
00010143000101430001014300010143
02FE00000000000000000000000FE02
Found 14443-A card :
ATQ=0400 UID=F425888A SAK=0800
Mifare 1k
00010143000101430001014300010143
03FD00000000000000000000000FD03
Found 14443-A card :
ATQ=0400 UID=F425888A SAK=0800
Mifare 1k
00010143000101430001014300010143
04FC00000000000000000000000FC04
```

6.5.2. The Mifare Application Directory (MAD)

Up to now, we've considered that data are always located at the same place in the card (static mapping to a defined block). This is all right in most situations, but sometimes we have to "share" the card along different kind of readers and applications, with flexibility and expandability for adding data in the future.

The Mifare Application Directory (MAD) concept defines how a dynamic card mapping can be implemented, using sector 0 (blocks 1 and 2) as a "directory" telling the reader where each data is located in the card.

For more information, read NXP's document "Mifare Application Directory"[14] or review case studies at www.mifare.net .

[14] At the time of writing, this document can be found online at http://www.nxp.com/products/identification/mifare/index.html#rel

6.5.3. *Working with one sector at once*

Block read (and write) functions work with 16 bytes of data.

Following functions are suitable to read (and write) one full sector at once :

-
  ```
  MifReadWriteSect(BOOL w, BYTE addr, BYTE data[],
                                  BYTE key_value[6])
  ```

-
  ```
  MifReadWriteSectK(BOOL w, BYTE addr, BYTE data[],
                                  BYTE key_ident)
  ```

Be careful that on Mifare 4k cards there're two different sector size (and therefore two different buffer size for the `data` parameter) : sectors 0 to 31 are made of 3 data blocks (+ 1 block for sector's trailer), i.e. 48 bytes, where sectors 32 to 39 are made of 15 data blocks (+ 1 block for sector's trailer), i.e. 240 bytes.

On Mifare 1k all sectors are 48 bytes.

7. WORKING WITH T=CL CARDS

ISO/IEC 14443 layer 4 is often named "T=CL protocol", after "T=0" and "T=1" smartcard asynchronous serial protocols. This implies that the application may exchange frames with the card, without any specific processing by the reader[15].

In our case, the application is physically running inside the reader, but this doesn't make a difference...

7.1. ENTERING ISO/IEC 14443 LAYER 4

7.1.1. *Type A*

14443-A T=CL card are recognized by bit 5 being set in SAK. To enable T=CL communication with the card, the reader shall send a "select" frame, to which the card answer with its Answer To Select (ATS).

More than one T=CL card may be selected at the same time by the application, using a short Card IDentifier (CID). In this chapter, we limit us to a single card. CID will be fixed to 0xFF ("CID not used" reserved value).

Here's the code for T=CL activation of a type A card :

```
if (iso3a_tag.sak[0] & 0x20)
{
  /* Card is T=CL compliant */
  rc = TclA_GetAts(0xFF, NULL, NULL);
}
```

Complete prototype of `TclA_GetAts` is :

`TclA_GetAts(BYTE cid, BYTE ats[], BYTE *atslen)`

You can use the `ats` parameter to retrieve card's Answer To Select (this is more or less the equivalent of the ATR of a T=0 or T=1 contact smartcard).

7.1.2. *Type B*

14443-B T=CL card are recognized by bit 0 of byte 9 being set in ATQ. To enable T=CL communication with the card, the reader shall send an "attrib" frame.

Once again, more than one T=CL card may be selected at the same time by the application, using a short Card IDentifier (CID). We limit us to CID = 0xFF ("CID not used" reserved value).

[15] Different from Mifare mode where the application relies on the reader –on the RC531, actually– to perform on-the-fly CRYPTO1 ciphering and de-ciphering.

Here's the code for T=CL activation of a type B card :

```
    if (iso3b_tag.atq[9] & 0x01)
    {
      /* Card is T=CL compliant */
      rc = TclB_Attrib(0xFF, iso3b_tag.atq);
    }
```

Type B "attrib" command (ATTRIB) is "addressed" to one specific card, so card's PUPI (4-first bytes of ATQ) must be provided to `TclB_Attrib`.

Type A "get ATS" command is "broadcasted", but only the currently selected card will accept the command, that's why `TclA_Halt` doesn't need card's UID.

7.2. EXCHANGING FRAMES WITH THE CARD

Once a T=CL card has been selected, exchanging frames with it is as easy as calling `Tcl_Exchange`, whatever the type of the card.

Here's the prototype :

```
Tcl_Exchange(BYTE cid,
             BYTE send_buffer[],
             WORD send_len,
             BYTE recv_buffer[],
             WORD *recv_len);
```

In our examples `cid` will be fixed to 0xFF. `send_len` (and `*recv_len`) are limited only by reader's memory and by card's in/out buffer –the second being often shorter than the first.

According to the OSI model, the Tcl_Exchange is on top of the stack, implementing a dialog *reader application* ←→ *card application*. Size of frames at this level is limited only by application specifications and available memory.

Lower layers in the stack may use shorter buffers ; in this case the application buffer must be split in one or more smaller frames.

Upon transmit, this is done automatically by **K531 INS library**, according to the size of receive buffer asserted by the card (value to be retrieved from ATS or ATQ).

In the other way, the reader is able to accept 256 bytes at once (maximum specified by ISO), but can also merge transparently the incoming frames split by a card having a too short transmit buffer.

7.2.1. ISO/IEC 7816 commands and APDUs

As ISO/IEC 14443-4 is the contactless equivalent of ISO/IEC 7816-3 T=1 protocol, most card manufacturers and/or card application designers implement ISO/IEC 7816-4 commands and T=1 formatted APDU in their card and/or applets[16].

Using ISO/IEC 7816 formalism, we can understand `send_buffer` as follow :

- Case 1 APDU

Offset	0	1	2	3
Item	CLA	INS	P1	P2

- Case 2 APDU

Offset	0	1	2	3	4
Item	CLA	INS	P1	P2	L_E

- Case 3 APDU

Offset	0	1	2	3	4	5 to `send_len-2`
Item	CLA	INS	P1	P2	L_C	Data

- Case 4 APDU

Offset	0	1	2	3	4	5 to `send_len–2`	`send_len–1`
Item	CLA	INS	P1	P2	L_C	Data	L_E

Using ISO/IEC 7816 formalism, we can understand `recv_buffer` as follow :

Offset	0 to `*recv_len-2`	`*recv_len–2`	`*recv_len–1`
Item	Data	SW1	SW2

Mapping of APDUs into T=CL frames is not clearly specified, and handling tge L_E byte appears to vary along applet developers.

We've seen some cards where L_E is ignored (case 2 being equivalent to case 1, case 4 being equivalent to case 3, both returning a variable length answer), some others where L_E must be removed in cases 2 and 4 (card always returns a variable length answer, and returns an error when L_E is provided), and some others working like T=0 cards (case 4 not allowed).

When working with a "7816-4 compliant" T=CL card, read carefully its documentation, looking for any precision regarding the mapping of APDUs.

Don't be surprise to receive more than L_E+2 bytes, and size `recv_buffer` in order to allow it. Keep in mind that L_E=0x00 can be understood either as "256 bytes" or as "any length up to 256 bytes".

[16] ISO/IEC 7816-4 "interindustry command for interchange" defines a basic command set for smartcards providing a file-system feature (directory and files selection, read and write into files) and secure communication.

7.2.2. *Other frame formats*

Some card manufacturers and/or card application designers choose to provide their own list of commands, with their own proprietary format, instead of using 7816-4 command set and APDU formalism.

For instance, NXP Desfire card use the following model :

- Application → card (`send_buffer`)

Offset	0	1 to `send_len–1`
Item	Command	Data

- Card → application (`recv_buffer`)

Offset	0	1 to `*recv_len–1`
Item	Status	SW1

> When working with such a proprietary protocol, pay a lot of attention to examples provided by card's developer. Try to prototype the application on PC with a desktop contactless reader. It will always be a gain of time, since debugging is virtually impossible in the K531.

7.2.3. *Closing communication correctly*

Once a card has entered 14443-4 layer, it remains active until you Deselect it, where it goes back into the Halted state (same as `IsoA_Halt` and `IsoB_Halt` when card is still at 14443-3 layer).

The Deselect function is :

```
Tcl_Deselect(BYTE cid)
```

> You must Deselect the card you're working with before trying to activate another card with the same CID, even if you assume that the card has been removed from the RF field.
>
> TclA_GetAts and TclB_Attrib will fail with error `TCL_CID_ACTIVE` if their `cid` parameter references a card that hasn't been Deselect.

7.3. DESFIRE EXAMPLE

As the Desfire cards supplied in the SDK are "blank", we limit us to the Desfire "GetVersion" command, which returns 28 bytes of data, split into 3 frames[17].

Reading data from a file on the card will use a different command sequence, but there's no difference in the method.

a. Source code

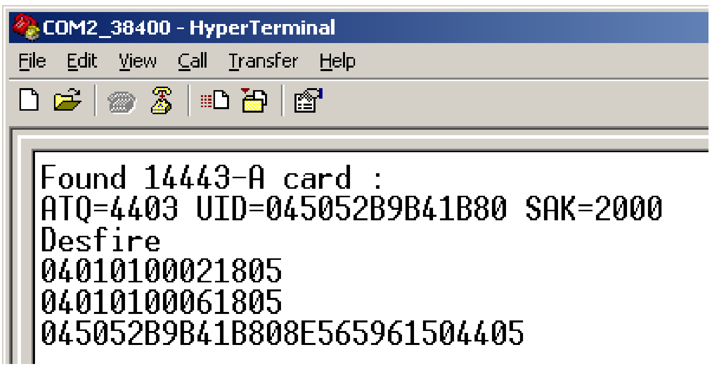
HelloWorld_7_3.c

```
     /* Is this a Desfire card  ? */
     if ((iso3a_tag.atq[1] == 0x03)
      && (iso3a_tag.atq[0] == 0x44))
     {
       BYTE send_buffer[1];
       BYTE recv_buffer[24];
       WORD recv_len;

       /* Yes ! */
       print_s("Desfire\r\n");

       /* Enter T=CL layer */
       rc = TclA_GetAts(0xFF, NULL, NULL);
       if (rc == MI_OK)
       {
         /* Send the GetVersion command */
         send_buffer[0] = 0x60;
         recv_len = sizeof(recv_buffer);
         rc = Tcl_Exchange(0xFF, send_buffer, 1,
                                 recv_buffer, &recv_len);
         if ((rc == MI_OK) && (recv_buffer[0] == 0xAF))
         {
           /* First exchange OK
              status is "OK, another frame to follow" */
           print_h(&recv_buffer[1], recv_len-1, FALSE);
           print_s(NULL);

           /* Ask for second frame */
           send_buffer[0] = 0xAF;
           recv_len = sizeof(recv_buffer);
           rc = Tcl_Exchange(0xFF, send_buffer, 1,
                                   recv_buffer, &recv_len);
           if ((rc == MI_OK) && (recv_buffer[0] == 0xAF))
           {
```

[17] Please refer to NXP's Desfire datasheet v3.1, paragraph 4.4.6, for details.

```
            /* Second exchange OK
               status is "OK, another frame to follow" */
            print_h(&recv_buffer[1], recv_len-1, FALSE);
            print_s(NULL);

            /* Ask for last frame */
            send_buffer[0] = 0xAF;
            recv_len = sizeof(recv_buffer);
            rc = Tcl_Exchange(0xFF, send_buffer, 1,
                                        recv_buffer, &recv_len);
            if ((rc == MI_OK) && (recv_buffer[0] == 0x00))
            {
              /* Third exchange OK
                 status is "OK, terminated" */
              print_h(&recv_buffer[1], recv_len-1, FALSE);
              print_s(NULL);
            }
          }
        }

        /* Deselect the Desfire card */
        Tcl_Deselect(0xFF);
      }
    } else
    {
      /* Not a Desfire card, halt it right now */
      IsoA_Halt();
    }
```

b. Output



```
Found 14443-A card :
ATQ=4403 UID=045052B9B41B80 SAK=2000
Desfire
04010100021805
04010100061805
045052B9B41B808E565961504405
```

A few explanations :

- First frame is "hardware information". It starts with Vendor ID = 0x04. That's NXP –formerly Philips Semiconductors–. Hardware release is 0.2 .
- Second frame is "software information". Again, it starts with Vendor ID = 0x04. Software release is 0.6 ("Desfire v6" card).

- Last frame contains UID, batch number, and production information. Observe that the 7-bytes UID also starts with 0x04, meaning that the card has been assigned its UID by NXP…

7.4. JAYCOS EXAMPLE

As the Jaycos cards supplied in the SDK are “blank”, we limit us to the Jaycos “GetATR” and “GetChipNumber” commands[18].

Reading data from a file on the card will use a different command sequence, but there’s no difference in the method.

a. *Source code*

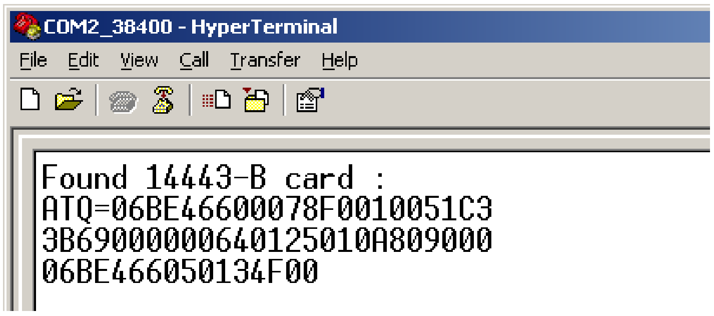
HelloWorld_7_4.c

```
        rc = IsoB_ActivateIdle(0x00);
        if (rc == MI_OK)
        {
          print_s("Found 14443-B card :\r\n");
          print_s("ATQ=");
          print_h(iso3b_tag.atq, 11, FALSE);
          print_s(NULL);

          /* Note : we can’t guess from ATQ only whether
             the card is a Jaycos or something else */

          /* Enter T=CL layer */
          rc = TclB_Attrib(0xFF, iso3b_tag.atq);
          if (rc == MI_OK)
          {
            BYTE send_buffer[5];
            BYTE recv_buffer[32];
            WORD recv_len;

            /* GetAtr APDU */
            send_buffer[0] = 0x80; /* CLA */
            send_buffer[1] = 0xEC; /* INS */
            send_buffer[2] = 0x00; /* P1  */
            send_buffer[3] = 0x00; /* P2  */
            send_buffer[4] = 0x0C; /* Le  */
            recv_len = sizeof(recv_buffer);
            rc = Tcl_Exchange(0xFF, send_buffer, 5,
                                    recv_buffer, &recv_len);
            if ((rc == MI_OK)
             && (recv_len >= 2)
             && (recv_buffer[recv_len-2] == 0x90)
```

[18] Please refer to Inseal’s Jaycos user guide v.AE, paragraphs 3.2.13 & 3.2.14, for details.

```
               && (recv_buffer[recv_len-1] == 0x00))
              {
                /* Exchange OK, SW = 90 00 */
                print_h(recv_buffer, recv_len-2, FALSE);
                print_s(NULL);

                /* Note : here we can consider that the card
                   is actually a Jaycos, because the command
                   CLA=0x80 INS=0xEC isn't standard */

                /* GetChipNumber APDU */
                send_buffer[0] = 0xB0; /* CLA */
                send_buffer[1] = 0xEE; /* INS */
                send_buffer[2] = 0x00; /* P1  */
                send_buffer[3] = 0x00; /* P2  */
                send_buffer[4] = 0x08; /* Le  */
                recv_len = sizeof(recv_buffer);
                rc = Tcl_Exchange(0xFF, send_buffer, 5,
                                        recv_buffer, &recv_len);
                if ((rc == MI_OK)
                 && (recv_len >= 2)
                 && (recv_buffer[recv_len-2] == 0x90)
                 && (recv_buffer[recv_len-1] == 0x00))
                {
                  /* Exchange OK, SW = 90 00 */
                  print_h(recv_buffer, recv_len-2, FALSE);
                  print_s(NULL);
                }

                /* Deselect the card */
                Tcl_Deselect(0xFF);
              }
            }
          }
```

b. Output



```
Found 14443-B card :
ATQ=06BE46600078F0010051C3
3B6900000064012501 0A809000
06BE466050134F00
```

A few explanations :

- First frame is the ATR of the card (Answer To Reset that is sent by the card when powered on by a "contact" smartcard reader). Observe that the ATR itself ends with 9000, so the GetAtr APDU response ends with 90009000.

- Second frame is the serial number of the card. It is 8 bytes long. Note that the 4 first bytes are used as PUPI in ATQ.

7.5. GOING FURTHER

7.5.1. Reading interesting data

Both examples studied here have read intrinsic data, that are available in the cards even when there're still blank. Of course a "real-life" reader application will have to access data stored in one or more files from the card.

The concept it always the same :

- Use layer 3 activation commands to discover the card(s) in the RF field,
- Try to recognize the card from its ATQ when possible,
- Enter layer 4 (T=CL),
- Select the file and fetch the data using APDUs or proprietary commands, depending on the card itself,
- Deselect the card when done.

Pay attention here to end-user experience in front of the reader. There's no such an unpleasant think as having to remove the card from the field and insert it back later, to overcome a communication error.

You must be really strict on error detection, and recognize the two different cases :

- Card communication error : keep trying silently until success or card removed,
- Card not correctly formatted, or invalid data read from the card : exit immediately, report a fatal error on LED and/or buzzer,

7.5.2. Trying to get secure…

Dialog between reader and T=CL card is not authenticated and not ciphered. If security is needed, it must be provided by an higher layer.

Here's a short list of possibilities in this domain :

- Implement a symmetric cipher algorithm in the reader[19], and use it for dynamic authentication[20] and secure communication,

[19] Due to a limited ROM size, implementing DES, 3-DES or AES in K531 will be really difficult, but a few tiny algorithms provide a decent security level (at least equivalent to Mifare CRYPTO1) and can be feat in the available ROM. Also consider switching to Pro-Active's K632 module (with embedded 3-DES and MD5 operators).

- Read a static signature from the card together with the data. Send card's UID, card's data and static signature to the host. Let host verify the signature[21],
- Generate a (pseudo)random number (nonce) in the reader, ask card to dynamically sign this number. Send reader's nonce, card's data and dynamic signature to the host. Let host verify the signature[22].

[20] In this case we'll also have to store the keys inside the reader, this is generally speaking not a good idea…

[21] This is a commonly used scheme, even on Mifare cards, built on RSA or Elliptic Curves asymmetric signature algorithms. Security relies on UID being actually unique, that may be discussed.

[22] This is also a commonly used scheme, typically by payment cards. The signature algorithm can be any kind of MAC computation (Message Authentication Code), built either on a symmetric cipher (DES, 3-DES, AES, …) or on an hash functions (MD5, SHA, …).

8. OTHER FEATURES

8.1. DRIVING LEDS

- Use function `set_leds` to configure the LED outputs.

The function accepts 3 parameters, for 3 LEDs : red, green and yellow. Red and green LEDs have their own output pins (17 & 18). Yellow LED is supposed to be bound to “user” pin (14).

Values for each parameter can be :

- `LED_OFF` : LED remains OFF (high level on the output pin),
- `LED_ON` : LED remains ON (low level on the output pin),
- `LED_FAST` : fast blinking,
- `LED_SLOW` : slow blinking,
- `LED_HEART` : “hear beat” blinking,
- `LED_DISABLED` : do not drive the LED output.

Never call `set_leds` with a value different than `LED_DISABLED` for yellow LED if you work with “user” pin either as general purpose I/O or as RS485 driver control line.

8.2. THE USER I/O PIN

- Use function `get_user` to configure the “user” pin (14) as input, and read its input level.
- Use function `set_user` to configure the “user” pin (14) as output and define its output level.

K531 INS library maps yellow LED to “user” pin.

Never call `set_leds` with a value different than `LED_DISABLED` for yellow LED if you work with “user” pin as general purpose I/O and not as yellow LED.

K531 INS library maps RS485 driver control to “user” pin.

Never call `set_rs485` if you work with “user” pin as general purpose I/O and not as RS485 driver control.

8.3. THE MODE I/O PIN

- Use function `get_mode` to configure the "mode" pin (16) as input, and read its input level.
- Use function `set_mode` to configure the "mode" pin (16) as output and define its output level.

On IWM-K531 reader, "mode" pin is an output that drives the buzzer.

8.4. WORKING WITH TIMERS

As seen in 3.2.2, the **K531 INS library** provides an easy way to implement timers, with a millisecond resolution.

If this example, we use 2 timers to perform some action on the LEDs :

- When a 14443-A card is found, we switch green LED ON and launch a fast blinker on yellow LED, both for 3 seconds. Red LED resume its "heart-beat" after 10s.
- When a 14443-B card is found, we do the exactly same, but with a slow blinker on yellow LED.

HelloWorld_8_4.c

```
static DWORD led_tmr_1, led_tmr_2;

void main(void)
{
  (...)

  for (;;)
  {
    (...)

    /* 14443-A lookup */
    rc = IsoA_ActivateAny();
    if (rc == MI_OK)
    {
      /* 14443-A LED sequence */
      set_leds(LED_OFF, LED_ON, LED_FAST);
      led_tmr_1 = timeout_init(3000);  /*  3000ms = 3s  */
      led_tmr_2 = timeout_init(10000); /* 10000ms = 10s */
    }

    (...)
```

```
        /* 14443-B lookup, AFI = 0 (any application) */
        rc = IsoB_ActivateAny(0x00);
        if (rc == MI_OK)
        {
          /* 14443-B LED sequence */
          set_leds(LED_OFF, LED_ON, LED_SLOW);
          led_tmr_1 = timeout_init(3000);  /*  3000ms = 3s  */
          led_tmr_2 = timeout_init(10000); /* 10000ms = 10s */
        }

        (...)

        /* Check if a timer has expired */
        if (timeout_expired(led_tmr_1))
        {
          /* Stop green & yellow LEDs */
          set_leds(LED_IGNORE, LED_OFF, LED_OFF);
          led_tmr_1 = timeout_kill();
        }
        if (timeout_expired(led_tmr_2))
        {
          /* Start "heart-beat" on red LED */
          set_leds(LED_HEART, LED_IGNORE, LED_IGNORE);
          led_tmr_2 = timeout_kill();
        }
      }
    }
```

8.5. DATACLOCK OUTPUT

K531 can be used to build a Dataclock reader. The Dataclock pins are multiplexed with the RX/TX pins of the serial line.

In this mode, pin “TX” (12) is the CLOCK line, and pin “RX” (11) the DATA line. Both lines are active low.

Since Dataclock outputs are multiplexed with UART, never call `print_s` or alike function, nor `serial_1_send_byte` when implementing a Dataclock reader.

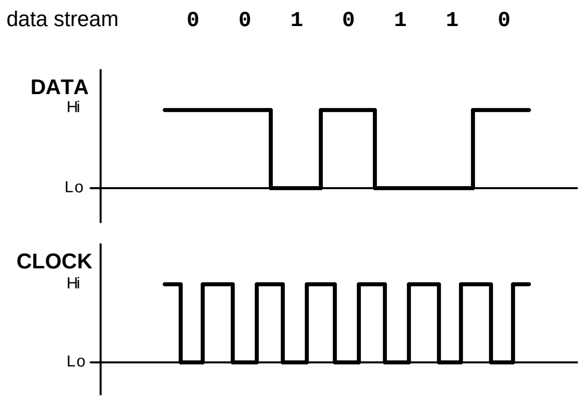
a. Dataclock functions

- Use function `dataclock_out` to send a decimal sequence on Dataclock outputs[23].
- Use `dataclock_out_dw` to send a DWORD (32 bits number) on Dataclock outputs[24].

b. Dataclock frame format

Both functions provides a valid ISO2 Dataclock frame, i.e. a frame starting with 16 dummy 0 bits for synchronisation, the Start Of Frame marker (`0xB`), and terminated by the End Of Frame marker (`0xF`) followed by the checksum (LRC). Inside the frame, a parity bit is added after each digit.

c. Dataclock data flow

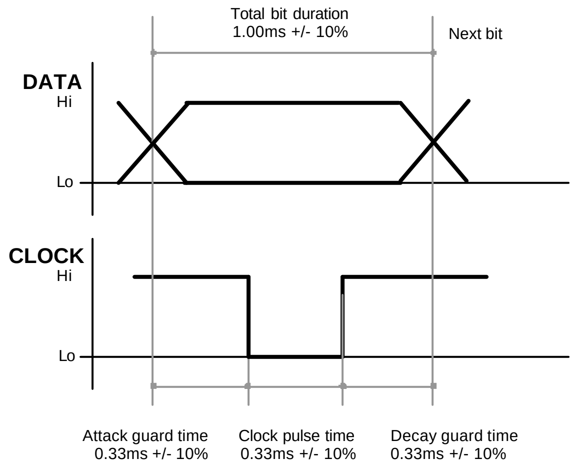


[23] The ISO2 Dataclock standard allows only BCD data (i.e. bytes where both nibbles are between 0 and 9). The sequence to be sent must be a valid BCD string :

- `dataclock_out("0123456789")` is correct,
- `dataclock_out("0123456789ABCDEF")` is incorrect (values `0xA` to `0xF` will be replaced by Dataclock separator `0xD`,
- `dataclock_out("GH...")` is forbidden and will produce an unspecified output.

[24] Size of output is exactly 10-decimal digits. For instance, DWORD value 0x001234AB (1193131 in decimal) will be transmitted as “0001193131”.

d. Dataclock bit format and timings



The default timings can be modified using function `dataclock_set_timing` (this function takes only one parameter ; the 3 times are always equals to keep the cyclic ratio at 1/3).

8.6. WIEGAND OUTPUT

K531 can be used to build a Wiegand reader. The Wiegand pins are multiplexed with the RX/TX pins of the serial line.

In this mode, pin “TX” (12) is the DATA1 line, and pin “RX” (11) the DATA0 line. Both lines are active low.

Since Wiegand outputs are multiplexed with UART, never call `print_s` or alike function, nor `serial_1_send_byte` when implementing a Wiegand reader.

a. Wiegand functions

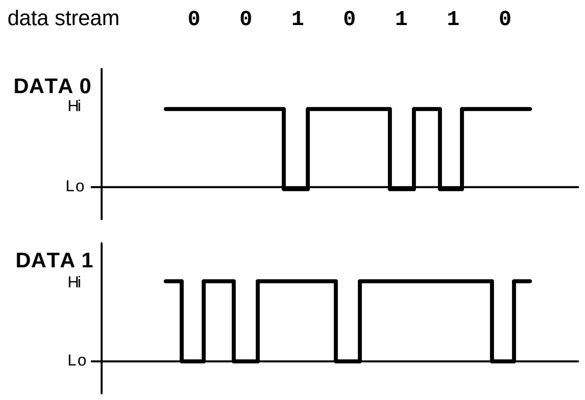
- Use function `wiegand_out` to send an hexadecimal sequence on Wiegand outputs[25].
- Use `wiegand_out_ex` to send an arbitrary buffer on Wiegand outputs[26].

b. Wiegand frame format

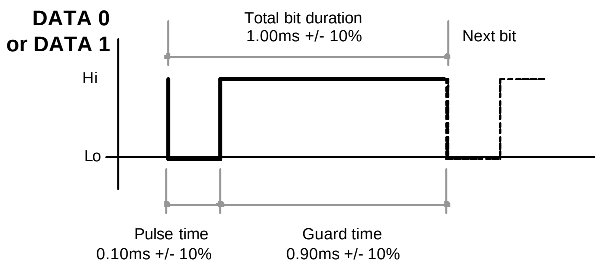
There's no frame marker, no parity bits, no checksum.

If you need a specific frame format, use `wiegand_out_ex` with a properly formatted buffer.

c. Wiegand data flow



d. Wiegand bit format and timings



The default timings can be modified using function `wiegand_set_timing`.

[25] The sequence to be sent must be a valid BCD or Hexadecimal string :

- `wiegand_out("0123456789")` is correct,
- `wiegand_out("0123456789ABCDEF")` is correct,
- `wiegand_out("GH...")` is forbidden and will produce an unspecified output.

[26] Calling `wiegand_out_ex("123ABZ", 6)` is exactly the same as calling `wiegand_out("31323341425A")`.

8.7. STORING NON-VOLATILE DATA

8.7.1. In the RC531

We've seen in "Mifare" chapter that access key can be stored in RC531's EEPROM. This chip also provides 4 bytes of "free" EEPROM that may be used to store a 32-bit value (4 bytes).

- Use function `PcdGetE2Data` to read this value,
- Use function `PcdSetE2Data` to write this value.

8.7.2. In R8C-25's data flash

R8C-25 features a 2kB flash memory dedicated to data storage. **K531 INS library** makes it available under the name "FEED" (Flash Emulating EEPROM for Data). The FEED can be seen as a list where persistent data can be inserted –and retrieve– by their line index (or item identifier).

Up to 254 items can be stored in the FEED. Each item can occupy any size between 1 and 32 bytes.

- Use function `feed_read` to read a, item from the list,
- Use function `feed_write` to insert or update an item in the list,
- Use function `feed_erase` to remove one item from the list.

9. IMPLEMENTING A "CONSOLE" ON THE SERIAL LINE

The code provided here is based on the "HelloWorld" sample. The key concept is to separate the receive interrupt handler (`serial_1_recv_callback`) from the command processor (in `main`). Do to so, we use a shared buffer (`recv_buffer`) and a shared boolean variable (`recv_ready`).

a. Updated `main` code

```
char recv_buffer[64];
volatile BOOL recv_ready;

void main(void)
{
  (...)

  for (;;)
  {
    (...)

    /* Something received on serial line ? */
    if (recv_ready && strlen(recv_buffer))
    {
      /* Dummy console processor,
         just echo back the command... */
      print_s("You've entered : \"");
      print_s(recv_buffer);
      print_s("\"\r\n");
      recv_buffer[0] = '\0';
      recv_ready = FALSE;
    }
  }
}
```

b. Updated `serial_1_recv_callback` code

```
void serial_1_recv_callback(BYTE r)
{
  int l = strlen(recv_buffer);

  if ((r == '\r') || (r == '\n'))
  {
    /* CR or LF -> ready to process the command */
    recv_ready = TRUE;
    /* Echo : send CR/LF */
    print_s(NULL);
  } else
  if (r == 0x08)
  {
    /* Backspace */
    if (l > 0)
      recv_buffer[--l] = '\0';
    print_b(r); print_b(' '); print_b(r);
  } else
  if (l < sizeof(recv_buffer)-1)
  {
    /* Enqueue in buffer */
    recv_buffer[l]   = r;
    recv_buffer[l+1] = '\0';
    /* Echo */
    print_b(r);
  }
}
```

10. OTHER SAMPLE INCLUDED IN THE SDK

a. Serno_Serial_Reader

This project is not really different from what we've seen in chapter 5. It implements a basic card lookup, and sends the information over the serial line.

There are 2 noticeable points anyway :

- We add a 1s (1000ms) delay between two consecutive outputs on serial line.

  This is a typical value in practical applications, where the receiver (access control device, cash-machine, …) is unable to process more than one "tag" at a time.

- We switch OFF the RF field between two consecutive lookups, with a 100ms interval between the pulses. The major consequence is to reduce average power needed by the device (and therefore the dissipated heat).

  The 100ms delay –hardly noticeable by end-user– is also a typical value, in some cases even 250ms are possible without significant impact on user's experience, where in other cases (long transactions, i.e. slow cards or lot of data to be read) 25 to 50ms may be better.

b. Serno_WiegandDataclock_Reader

Same as above, but with a Wiegand or Dataclock output. In this code the output mode is defined at compile time (`BOOL output_wiegand`), but it is easy to move it to a persistent configuration area, either RC531's EEPROM (§ 8.7.1) or in the FEED (§ 8.7.2).

Observe how we translate the 4-byte card ID to a 10-digit decimal number when working in Dataclock mode, while we send it without prior translation in Wiegand mode.

👍	This really simple code is at the basis of Pro-Active IWM-K531. We've only added a "configuration card" handler and store the settings in RC531's EEPROM.

c. Mifare_Serial_Reader

This is a practical implementation of what we've seen in chapter 6. Once again, configuration data (`BYTE address_on_tag`) should be moved to a persistent configuration area.

Interesting point : when we fail to read the card, we try again immediately instead of waiting 100ms.

d. Mifare_WiegandDataclock_Reader

Same as above, but with a Wiegand or Dataclock output.

e. Mifare_Serial_Encoder

This is an “improvement” of paragraph c, but now we write something in the card each time we see it. The serial line accepts two commands : “E” to erase the cards (write zeroes instead of data), and “W” to go back to write mode.

DISCLAIMER

COPYRIGHT NOTICE

EDITOR'S INFORMATION

Published by **Pro-Active SAS**, 13, voie La Cardon 91120 Palaiseau – France

R.C.S. EVRY B 429 665 482 - APE 722 Z

For more information, please contact us at info@pro-active.fr .