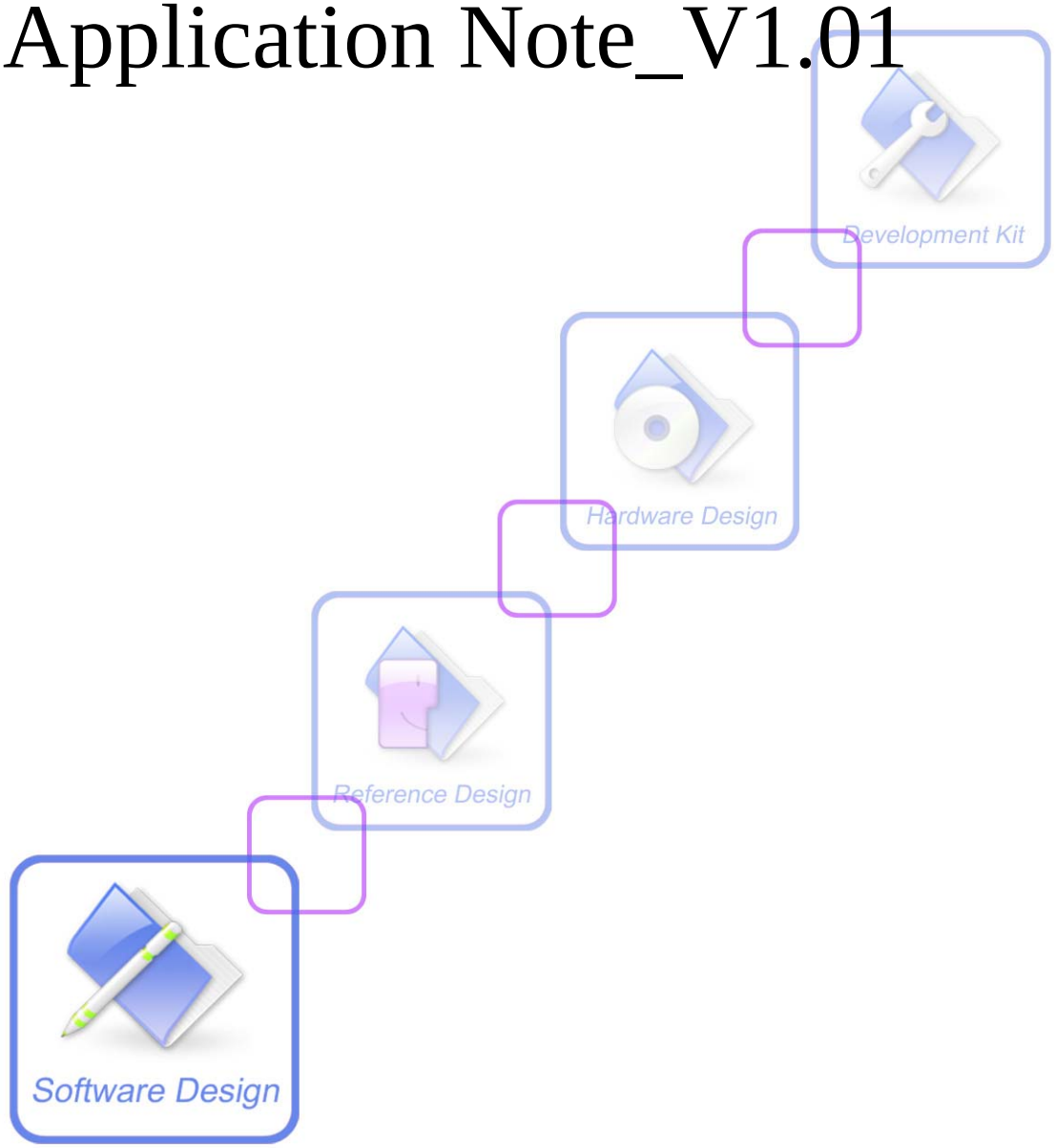
A company of SIM Tech

SIM900_Embedded AT® Application Note_V1.01



Document Title:	SIM900_Embedded AT® Application Note
Version:	1.01
Date:	2011-12-30
Status:	Released
Document Control ID:	SIM900_Embedded AT® Application Note_V1.01

General Notes

SIMCom offers this information as a service to its customers, to support application and engineering efforts that use the products designed by SIMCom. The information provided is based upon requirements specifically provided to SIMCom by the customers. SIMCom has not undertaken any independent search for additional relevant information, including any information that may be in the customer's possession. Furthermore, system validation of this product designed by SIMCom within a larger electronic system remains the responsibility of the customer or the customer's system integrator. All specifications supplied herein are subject to change.

Copyright

Contents

Version history

Date	Version	Description of change	Author
2010-09-01	V1.00	Origin	MXN
2010-03-31	V1.01	1. Added SIM900 embedded at. 2. Added some useful functions.	MXN

1. Introduction

1.1 Purpose

Based on ARM926EJ_S core, SIM900 runs at 156 MHz, and has redundant MIPS to run programs other than telecommunication protocols. Embedded AT is for fully utilizing Sim900 resources, providing interfaces to move some external MCU functions into itself, so as to save customer's cost. The programming idea of Embedded AT is to think from MCU side and to be consistent with the MCU programming style.

1.2 Coding style

The function name of EMBEDDED AT consists of two parts， one is the file name index part, and the other is the function number of the file. For example, "ebdat4_01GetMemory", 4 is the file name index part, and 01 is the function number of the file. It is very easy for the user or the SIMCom developers to trace problems this way.

1.3 References

SIM900_ATC_V1.05

1.4 Glossary

Glossary	Description
Embedded Application API	Software interfaces developed by SIMCom and open to licensed embedded application developers. The APIs include audio API, FCM API, flash API, system API, periphery API, STDLIB API, timer API and debug API
Embedded Application	User created application that utilizes Embedded API functions to interact with SIMCom core software, only to run on a SIMCom product
SIMCom Core System	The Core system released by SIMCom, which includes the core binary file and SIMCom library
EVENT	Capitalized EVENT notion used in this document represents specified system EVENT in embedded application. See Chapter 3 EVENT for EVENT definition

1.5 Abbreviations

Abbreviation	Description
API	Application Programming Interface
CPU	Central Processing Unit
FCM	Flow Control Manager
KB	Kilobyte
OS	Operating System
PDU	Protocol Data Unit
RAM	Random-Access Memory
ROM	Read-Only Memory
RTK	Real-Time Kernel
SMS	Short Message Services
SDK	Software Development Kit

This document describes the important points to which attention should be paid by the clients when they design their applications. As SIM900 can be integrated into a wide range of applications, the application notes are described in great detail.

This document can help user to quickly understand SIM900 interface, specifications, electrical and mechanical details. With the help of this document and other SIM900 application notes, users can use SIM900 module to design and set-up mobile applications quickly.

2 Description

2.1 Software Architecture

2.1.1 Software Organization

The software architecture of the Embedded AT facility is shown below:

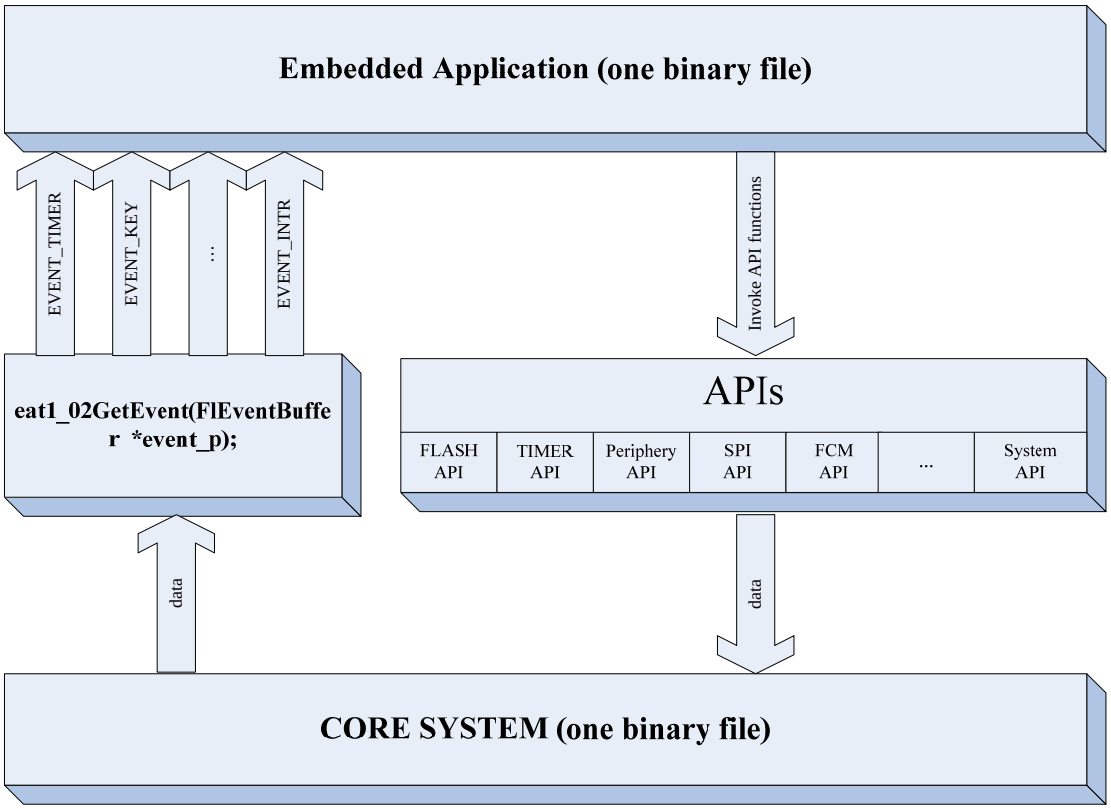


Figure 1: General software architecture

Information flow

When the module passes messages from the core system to the embedded application, the eat1_02GetEvent function catches EVENT and then alters the embedded application with categorized EVENT type. This approach allows the embedded application to channel different incoming data appropriately.

When the module sends messages to the core system, developer simply invokes API functions tailored for the appropriate purposes, and the rest is taken care of by the API functions, until the feedback message is received by the application through eat1_02GetEvent.

Message flows in this cyclical fashion, isolating the developer's application from accessing core variables and stacks. Through this design, Embedded AT masks and protects the core image from developer's application, allows abstract and safe access to the core system.

2.1.2 Resource supplied by SIMCom

Resources supplied by SIMCom are as following:

- 1M bytes code
- 1M bytes RAM
- 1M bytes memory which user can store the data in.
- 24 GPIOs
- 10 timers and one tick is equal to 9.23 ms.
- 1 SPI
- 1 Display interface
- 2 PWM
- 1 debug port
- 1 UART
- System API and standard library API

2.1.3 Software Supplied by SIMCom

The Softwares supplied by SIMCom are as following:

- One set of header files (.h) which define the Embedded API functions
- Source code samples
- SIMCom core software, which is a binary file
- Image downloading tools

2.2 Minimum Embedded Application Code

The following code is an example of the minimum embedded application code.

```
/*main function */
void fl_entry()
{
    bool keepGoing = TRUE;
    while (keepGoing == TRUE)
    {
        eat1_02GetEvent (&flEventBuffer); /*get event from SIMCom Core software*/
        switch(flEventBuffer.eventTyp)
            {
                ……
                default:
                break;
            }
    }

}
```

fl_entry is the main entrance function for embedded application, see next section for details.

eat1_02GetEvent is a system interface that receives EVENT from core software. See 4.2.1 for details.

2.3 fl_entry()

fl_entry() is the entrance function of embedded application, it works almost as main() in standard C application. Embedded application quits upon exiting fl_entry(). Above example uses a while statement to keep the application alive until the application developer ends it by setting keepGoing = FALSE;

```
while (keepGoing == TRUE) /*the while statement to keep embedded application alive*/
```

2.4 Embedded AT Memory resources

The Embedded software runs within real time kernel task: application developers must work with pre-defined size, which is 10K bytes, of the customer's application calling stack. Please note that the total size of local variables which user defines cannot exceed 10K bytes.
SIMCom Core Software and Embedded Application manage their own RAM areas. Access from one of these programs to another's RAM area is prohibited and will cause fatal error.
Global variables, call stack and dynamic memory are all part of the RAM allocated to the Embedded Application.

3 EVENT

EVENT is wrapped in structure FlEventBuffer, through which the core system communicates with the embedded applications. Only through eat1_02GetEvent(&flEventBuffer), EVENTs can be passed from the core system to the embedded applications. Structure FlEventBuffer consists of two parts. one is the event type, which defines the type of the EVENT, and the other is the event data.

```
typedef struct FlSignalBufferTag
{
    FlEventType   eventTyp;
    EventData     eventData;
}FlEventBuffer;
```

3.1 EVENT Type

3.1.1 FlEventType

EVENTs are categorized as following:

```
typedef enum FlEventTypeTag
{
    EVENT_NULL = 0,
    EVENT_INTR,
    EVENT_KEY,
    EVENT_UARTDATA,
    EVENT_MODEMDATA,
    EVENT_TIMER,
    EVENT_SERIALSTATUS,
    EVENT_SOCKET,
    EVENT_MAX = 0xFF
}FlEventType;
```

3.1.2 EVENT_INTR

The event is triggered by an interrupt signal which the embedded application receives from the core. Interrupt signals are generated by the interrupt pins, for details on the interrupt pins please refer to Section 4.4 Periphery API. Once a level change occurs on one of the interrupt pins, this event is received by the embedded application.

3.1.3 EVENT_KEY

The event is triggered when a key status is changed, which is a key press or a key release. By default there is a predefined keypad of five columns and five rows. When one of the key status (assume above mentioned pins have not been configured for other uses, for pin configuration refer to section 4.4) has been changed, the event **EVENT_KEY** is received by the embedded application.

3.1.4 EVENT_UARTDATA

The event is triggered when input data from serial port or trace port are received by SIMCom core firmware.

Important Remark:

In order to receive data from UART port in user's embedded application, **ebdat9_04SetUartdataToFL(TRUE)** has to be set. By default it is set to **ebdat9_04SetUartdataToFL(FALSE)**, the data received from UART port will be sent directly to the SIMCom core software. In default mode, embedded application will not receive data from the UART port and **EVENT_UARTDATA will never** be triggered.

3.1.5 EVENT_MODEMDATA

The event is triggered when modem data are sent to serial port, for instance, when the serial port

receives an AT command response.

Important Remarks:

- The same situation in EVENT_UARTDATA applies here too, the function ebdat9_03SetModemdataToFL(TRUE) has to be set (default is FALSE) before embedded application can capture SIMCom core outputs, such as OK or ERROR returned by AT commands.
- AT+CRWP is the exceptional case, despite of ebdat9_03SetModemdataToFL setting, embedded application will always receive it. For more details on AT+CRWP refer to Chapter 5 AT+CRWP.

3.1.6 EVENT_TIMER

The event is triggered when a timer expires. Timer can be stopped before it expires. For more details on timers, refer to TIMER API section.

3.1.7 EVENT_SERIALSTATUS

The event is triggered when serial port status has been changed, the status can be CTS, DCD, RI (ringing), DSR, DTR, and RTS.

3.1.8 EVENT_SOCKET

This event will be triggered when using SOCKET API of Embedded AT, including GPRS setup and release, setting up or closing TCP/UDP, sending or receiving data via TCP/UDP, etc.

3.1.9 Example

The following code skeleton demonstrates how events are captured in embedded applications:

```
void fl_entry() /*customer entrance*/
{          /* some code here*/
          switch(flEventBuffer.eventTyp) /* deal with signal associated to the signal type*/
            {
             case EVENT_INR:
             break;
             case EVENT_UARTDATA:
             break;
             case EVENT_MODEMDATA:
             break;
             case EVENT_KEY:
             break;
             case EVENT_TIMER:
             break;
```

```
            …
            default:
            break;
        }
      }
}
```

3.2 EVENT Data

3.2.1 EventData

Each EVENT type has its corresponding EVENT data.

```
typedef union EventDataTag
{
    TIMER_EVT               timer_evt;
    KEY_EVT                 key_evt;
    UARTDATA_EVT            uartdata_evt;
    MODEMDATA_EVT           modemdata_evt;
    INTR_EVT                intr_evt;
    SERIALSTATUS_EVT        serialstatus_evt;
    SOCKETEVENT_EVT         socket_evt;
}EventData;
```

Note EventData is not like EventType, EventData is a union, and each data type has its own structure, which will be detailed in the following sections.

3.2.2 TIMER_EVT

```
typedef struct TIMER_EVTTag
{
    u16   timer_id;
    u32   interval;
}TIMER_EVT;
```

timer_id: ID of the timer that has expired.
interval: The time elapsed before the timer expired. It is measured in Kernel ticks.

3.2.3 KEY_EVT

```
typedef struct KEY_EVTTag
{
    u16      key_val;
    bool     isPressed;
}KEY_EVT;
```

key_val: The value of the key that triggers the event.

isPressed: Whether the key is pressed. If it is 0, key is released, otherwise it is pressed.

3.2.4 UARTDATA_EVT

```
typedef struct UartData_EVTTag
{
    u8                  length;
    u8                  data[EVENT_MAX_DATA];
    FlUartDataType      type;
} UARTDATA_EVT;
```

length: The length of the data being transported.

data: The actual data, which is 255 bytes long maximum.

type: The type of the data, FlUartDataType type, see below for definition of FlUartDataType.

FLUartDataType

```
typedef enum UARTDATA_TYPETAG
{
    DATA_SERIAL = 0,
    DATA_DEBUG,
    MODEMDATA_MAX
} FlModemDataType;
```

3.2.4.1 DATA_SERIAL

Indicate the type of data which are received from serial port.

3.2.4.2 DATA_DEBUG

Indicate the type of data which are received from the trace port.

3.2.5 MODEMDATA_EVT

```
Typedef struct ModemData_EVTTag
{
    u8                      length;
    u8                      data[EVENT_MAX_DATA];
    FlModemDataType         type;
    u32                     atCommandIndex;
} MODEMDATA_EVT;
```

length: The length of the data being transported.

data: The actual data, which is 255 bytes long maximum.

type: The type of the data, FlDataModemType types, see below for definition of FlModemDataType.

FLModemDataType

```
typedef enum MODEMDATA_TYPETAG
{
    MODEM_CMD=0,
    MODEM_DATA,
    MODEM_CRWP,
    MODEMDATA_MAX
}FlModemDataType;
```

atCommandIndex:

When the customer defines an AT command and the AT command is received from the serial port, the EVENT_MODEMDATA will be triggered. The “atCommandIndex” is the AT command index which is defined by the customer.

3.2.5.1 MODEM_CMD

AT command data type. Refer to *Appendix B*.

3.2.5.2 MODEM_DATA

In data mode, this event will be triggered by any data, such as PPP data, CSD data or TCP data.

3.2.5.3 MODEM_CRWP

CRWP data type is the data type used in AT+CRWP command. For more information on +CRWP command, refer to Chapter 5.

3.2.6 INTR_EVT

```
typedef struct INTR_EVTTag
{
    flPinName      pinName;
    bool           gpioState;
}INTR_EVT;
```

pinName: Name of the pins on SIMCom modules.

gpioState: The status of the pin, if it is 0, a falling edge or low level interrupt happens. If it is 1, a rising edge or high level interrupt happens.

3.2.7 SERIALSTATUS_EVT

```
typedef enum SERIAL_BITTAG
{
    RI=0,
    DCD,
    DSR,
    DTR,
    CTS,
    RTS
}FlSerialBit;
typedef struct SERIALSTATUS_EVTTag
{
    u8 currentVal;
    FlSerialBit sbit;
}SERIALSTATUS_EVT;
```

currentVal: Serial port data. If it is 1, the pin on the serial port is high level. If it is 0, the pin on the serial port is low level.

sbit: Serial port status

3.2.8 SOCKETEVENT_EVT

```
typedef enum FlSocketEventTypeTag
{
    FL_SOCKET_CONNECT,
    FL_SOCKET_SEND,
    FL_SOCKET_RECV,
    FL_SOCKET_CLOSE,
    FL_SOCKET_REMOTE_CLOSE,
    FL_SOCKET_TCP_SERVER_START,
    FL_SOCKET_TCP_SERVER_CONNECT,
    FL_SOCKET_TCP_SERVER_STOP,
    FL_SOCKET_GPRS_ACTIVE,
    FL_SOCKET_GPRS_DEACTIVE,
    FL_SOCKET_MAX
}FlSocketEventType;

typedef struct SOCKET_EVTTag
{
    FlSocketEventType type;
    u32                 socketId;
    u32                 bsdResult;
}SOCKET_EVT;
```

type: Different types of socket event.

socketId: Represents different socket connections, it will be set to 0XFFFFFFFF when it is FL_SOCKET_GPRS_ACTIVE and FL_SOCKET_GPRS_DEACTIVE.

bsdResult: Represents different results of socket events, success or failure, or represents data length of sending and receiving.

3.2.9 Examples

```
Case EVENT_TIMER: /*deal with the timer event*/
        if(flEventBuffer.sig_p.timer_evt.timer_id == timerDemo.timerId)
        {
            /*deal with the timerDemo's event*/
            ebdat9_02SendToSerialPort("the timerDemo is coming!\x0d",25);
            /*show string on terminal window*/
        }
    break;
```

In this example, timerDemo.timerId is compared with the expired timer's ID, if timerDemo is expired, the embedded application will send "the timerDemo is coming!" to the serial port.

4 API

This chapter categorizes API functions and describes their usages, including function prototype, parameters, and their return values.

4.1 Data Types

File \flinc\fl_typ.h declares all the data types used in SIMCom Embedded AT.

```
typedef unsigned    char        bool;    /*TURE or FALSE*/
typedef unsigned    char        u8;
#define gu8 u8 __align(4)
typedef signed      char        s8;
#define gs8 s8 __align(4)
typedef             char        ascii;
#define gascii ascii __align(4)
typedef unsigned    short       u16;
typedef             short       s16;
typedef unsigned    int         u32;
typedef             int         s32;
typedef unsigned    int         ticks;
```

Note: fl_typ.h does not need to be included every time, since it is included in fl_interface.h, and when the char or byte buffer are defined as global variables, user should use "gu8", gs8 and gascii otherwise, abrupt reset may occur.

4.2 System API

File \flinc\fl_interface.h declares system-related APIs. These functions are essential to any customer applications, he head file needs to be included. User can use these functions to allocate a memory or to free the memory.

4.2.1 eat1_02GetEvent

The eat1_02GetEvent function gets system EVENTs from the core software. When there is no event in customer task's event queue, the task is in the waiting status.

- **Prototype**

```
void eat1_02GetEvent(FlEventBuffer    *event_p);
```

- **Parameters**

event_p: A pointer to a particular FlEventBuffer, refer to Chapter 3 for details.
EVENT for FlEventBuffer structure.

The following code is an example of how to create a signal buffer, and listen to incoming signals using **eat1_02GetEvent** function.

```
void fl_entry()
{
    bool keepGoing = TRUE;
    FlEventBuffer      flEventBuffer;
    while (keepGoing == TRUE)
    {
        /*get EVENT from SIMCom Core software*/
        eat1_02GetEvent(&flEventBuffer);
        switch(flEventBuffer.eventTyp)
        {
            ……
        }
    }
}
```

4.2.2 ebdat4_01GetMemory

The ebdat4_01GetMemory function will allocate memory from the memory pool.
Note:
The maximum size of the memory that user can allocate is 8K bytes. If user allocates the memory with size larger than 8K bytes, it will return NULL which means memory allocation is failed. It is better to define a global buffer than to allocate a memory, when the size of the buffer is larger than 100 bytes.

- **Prototype**

```
void *ebdat4_01GetMemory(u16 Size);
```

- **Parameters**

Size: The size of memory which will be allocated.

- **Return values**

It returns the address of the allocated memory. If it returns NULL, it means that the memory

allocation is failed.

4.2.3 ebdat4_02FreeMemory

The ebdat4_02FreeMemory function frees the memory which was allocated earlier. Note that user cannot free a NULL pointer.

- **Prototype**

```
bool ebdat4_02FreeMemory (void *Ptr);
```

- **Parameters**

Ptr: The address of the allocated memory

- **Return values**

TRUE (1): If allocated memory is freed.
FALSE (0): Is returned otherwise.

4.2.4 ebdat4_03Reset

The ebdat4_03Reset function resets the system. If user wants to reset the module, user can use this function. Use this function cautiously. It is not recommended to use this function generally.

- **Prototype**

```
void ebdat4_03Reset(void);
```

4.2.5 ebdat4_04Wdtkick

The ebdat4_04Wdtkick function kicks the watch dog. Call this function cautiously, only call it when the execution time of customer's code exceeds watchdog's reset time.

- **Prototype**

```
void ebdat4_04Wdtkick(void);
```

4.2.6 ebdat4_05PowerDown

The ebdat4_05PowerDown function powers down the system. It has the same effect as the AT command "AT+CPOWD=1". When the system is powered down successfully, "NORMAL POWER DOWN" will be sent to the serial port.

- **Prototype**

```
void ebdat4_05PowerDown(void);
```

4.2.7 eat1_09UpdateEmbeddedAp

See 4.11 Updating Embedded Application.

4.2.8 ebdat6_17DisablePowerOffKey

The ebdat6_17DisablePowerOffKey function configures the power key as a normal key. If the power key is pressed, EVENT_KEY will be triggered, and the value of key_val will be 0x0000. In default mode, the power key is enabled.

- **Prototype**

```
void ebdat6_17DisablePowerOffKey(void);
```

4.2.9 ebdat6_18EnablePowerOffKey

The ebdat6_18EnablePowerOffKey function enables the power key. When this function is called, the power key will be set to power off key. In default mode, the power key is enabled.

- **Prototype**

```
void ebdat6_18EnablePowerOffKey(void);
```

4.3 FLASH API

User can use these interfaces to store, read or delete the data in the flash. User can also use these interfaces to get the data length in the flash and the free size of the flash. In order to use these interfaces the header file fl_flash.h must be included. The length of the data written in flash cannot exceed 8K bytes.

Note:

1. ***Flash ID number cannot exceed 60000. Before writing the data to the flash, a buffer should be defined. When the buffer is defined, "gu8" should be used as "gu8 g_writeBuffer[8*1024];".***
2. ***If the customer wants to use updated Embedded Application, ebdat3_03FlashWriteData and ebdat3_04FlashReadData should be used.***

4.3.1 ebdat3_05FlashGetLen

Get the length of a specific flash.

- **Prototype**

```
s32 ebdat3_05FlashGetLen(u16 ID,u16* len);
```

- **Parameters**

ID: ID of the flash. The value of ID must be less than 60000, otherwise it will return FL_RET_ERR_PARAM.
len: The length of the flash area defined by its ID.

- **Return values**

FL_OK: Get the length successfully.
FL_RET_ERR_PARAM: Incorrect Incorrect parameter.
FL_RET_ERR_FATAL: If a fatal error occurred.

4.3.2 ebdat3_06FlashDelete

The ebdat3_06FlashDelete function deletes a region of the flash defined by an ID

- **Prototype**

```
s32 ebdat3_06FlashDelete(u16 ID);
```

- **Parameters**

ID: The ID of the flash object to be deleted. The value of ID cannot exceed 60000, otherwise it will return FL_RET_ERR_PARAM.

- **Return values**

FL_OK: The region of the flash is deleted successfully.
FL_RET_ERR_PARAM: Incorrect parameter**.**
FL_RET_ERR_FATAL: If a fatal error occurred.

4.3.3 ebdat3_07FlashGetFreeSize

The ebdat3_07FlashGetFreeSize function gets the free size on the flash which users can allocate.

- **Prototype**

s32 ebdat3_07FlashGetFreeSize(u32 *freeSize);

- **Parameters**

***freeSize:** Returns the free size of the flash.

- **Return values**

FL_OK: On success.
FL_RET_ERR_FATAL: If a fatal error occurred.

4.3.4 ebdat3_03FlashWriteData

The ebdat3_03FlashWriteData function writes data to a flash object of a given ID. The size of the flash object is defined in "**len**" parameter.

- **Prototype**

s32 ebdat3_03FlashWriteData(u16 ID, u16 len, u8 * data);

- **Parameters**

ID: The ID of the flash object to be written. The value of ID cannot exceed 60000, otherwise it will return FL_RET_ERR_PARAM.
len: The length of the flash object to be written. It cannot exceed 8K bytes otherwise it will return FL_RET_ERR_PARAM.
data: The string to be written into the flash object. It should not be NULL otherwise it will return FL_RET_ERR_PARAM.

- **Return values**

FL_OK: Write data to flash successfully.
FL_RET_ERR_PARAM: Incorrect parameter.
FL_RET_ERR_FATAL: If a fatal error occurred.

4.3.5 ebdat3_04FlashReadData

The ebdat3_04FlashReadData function reads data from a specific flash object with a given ID.

- **Prototype**

s32 ebdat3_04FlashReadData(u16 ID, u16 len, u8 * data);

- **Parameters**

ID: The ID of the flash object to be read. It cannot exceed 60000, otherwise FL_RET_ERR_PARAM will be returned.
len: The length of the flash object to be read. It cannot exceed 8K bytes or the size of the object user wants to read, otherwise FL_RET_ERR_PARAM will be returned.
data: The data allocated to store the flash object. It should not be NULL, otherwise FL_RET_ERR_PARAM will be returned.

- **Return values**

FL_OK: Read data from flash successfully.
FL_RET_ERR_PARAM: Incorrect parameter.
FL_RET_ERR_FATAL: If a fatal error occurred.

4.3.6 ebdat3_08FlashFileRead

The ebdat3_08FlashFileRead function allows customer to read a file from the file system in the module. But note that the filename cannot include its path.

- **Prototype**

```
s32 ebdat3_08FlashFileRead(u16 len, u8* data, u8* fileName, u16 position);
```

- **Parameters**

len: the length of the file which will be read to the module.
data: the data of file which will be read to the module.
fileName: the file name which will be read to the module.
position: the position of the file where it starts to read from. It is similar to the seek function.

- **Return values**

FL_OK: Read a file from flash successfully.
FL_RET_ERR_PARAM: Incorrect parameter.
FL_RET_ERR_FATAL: If a fatal error occurred.

4.3.7 ebdat3_09FlashFileWrite

The ebdat3_09FlashFileWrite function allows the customer to write a file to the file system in the module. But note that the file name cannot include its path.

- **Prototype**

```
s32 ebdat3_09FlashFileWrite(u16 len, u8* data, u8* fileName, FlFileOperationMode mode);
```

- **Parameters**

len: the length of the file which will be written to the module.
data: the data of the file which will be written to the module.
fileName: the file name which will be written to the module.
mode: the mode which defines how the customer writes a file into module.
FlFileOperationMode

```
typedef enum FlFileOperationModeTag
{
    FL_FILE_FROM_BEGINNING,/*create a new file, the previous one will be deleted.*/
    FL_FILE_FROM_END, /*write the data to the end of the previous file.*/
    FL_NUM_FILE_OPERATION_MODE
}FlFileOperationMode;
```

- **Return values**

FL_OK: write a file into flash successfully.
FL_RET_ERR_PARAM: Incorrect parameter.
FL_RET_ERR_FATAL: If a fatal error occurred.

4.3.8 ebdat3_10FlashFileDelete

The ebdat3_10FlashFileDelete function allows the customer to delete a file in the file system in the module. But note that the file name cannot include its path.

- **Prototype**

```
s32 ebdat3_10FlashFileDelete(u8* fileName);
```

- **Parameters**

fileName: the file name which will be deleted from the module.

- **Return values**

FL_OK: delete the file in flash successfully.
FL_RET_ERR_PARAM: Incorrect parameter.
FL_RET_ERR_FATAL: If a fatal error occurred.

4.3.9 ebdat3_11FlashFileGetLen

- **Prototype**

```
s32 ebdat3_11FlashFileGetLen(u8* fileName,u16*    length);
```

- **Parameters**

fileName: the file name which will be deleted from the module.
length: return the file length.

- **Return values**

FL_OK: write data into flash successfully.
FL_RET_ERR_PARAM: Incorrect parameter.
FL_RET_ERR_FATAL: If a fatal error occurred.

4.4 Periphery API

File fl_Periphery.h must be included before following functions are called. In this part, user can use these interfaces to control the periphery of the module such as the keypad, gpio, spi, interrupt, etc..

4.4.1 Module Pins

This section describes the pins of SIMCom modules. It includes the reference names used in the program code, and their operation mode.

4.4.1.1 FlPinName

FLPinName lists pin names, and their available operation mode.
For SIM900 see *Appendix A*:SIM900 FlPinName enum.

```
Typedef enum FlPinNameTag
{
        FL_PIN_3, /*Note:This pin cannot be used as GPIO. It is reserved.*/
        FL_PIN_4,
        FL_PIN_5,
        FL_PIN_6,
        FL_PIN_11,
        FL_PIN_12,
        FL_PIN_13,
        …
        …
        …
        FL_PIN_66,
        FL_PIN_67,
        FL_PIN_68,
        FL_PIN_MAX
} FlPinName;
```

4.4.1.2 FlPinMode

FlPinMode defines the pin mode. Each pin can only be subscribed to one purpose at any given time. There is no default mode for unused pins.

```
typedef enum FlPinModeTag
{
        FL_PIN_MODE_UNUSED,
        FL_PIN_MODE_DEFAULT,
        FL_PIN_MODE_MULTI,
        FL_PIN_MODE_GPIO,
        FL_PIN_MODE_I2C
} FlPinMode;
```

4.4.2 Periphery functions

This section describes API functions that deal with general pin mode manipulation.

4.4.2.1 ebdat6_08pinConfigureToUnused

The ebdat6_08pinConfigureToUnused function unsubscribes the named pins and configures the pin mode to be **FL_PIN_MODE_UNUSED**. Before the pin is configured as a GPIO, this function must be called first.

- **Prototype**

```
s32 ebdat6_08pinConfigureToUnused(FlPinName pinName);
```

- **Parameters**

pinName: The name of the pin to be set to **FL_PIN_MODE_UNUSED** status. Note that FL_PIN_3 cannot be configured as a GPIO, as it is reserved.

- **Return values**

Fl_OK: Set the pin to **FL_PIN_MODE_UNUSED** status successfully.
FL_RET_ERR_PARAM: Incorrect parameter
FL_RET_ERR_BAD_STATE: If the pin's status is unexpected
Note:

- ***It is important to unsubscribe pins from their current usage before assigning them to another purpose. Otherwise FL_RET_ERR_BAD_STATE will be returned.***
- ***All the keypad pins will be unassigned if one of the pins is unsubscribed.***

4.4.2.2 ebdat6_06QueryPinMode

The ebdat6_06QueryPinMode function queries the named pin's operation mode.

- **Prototype**

```
s32 ebdat6_06QueryPinMode(FlPinName pinName,
                          FlPinMode *pinMode_p,
                          FlGpioDirection *isOutputDir_p);
```

- **Parameters**

pinName: The name of the pin to be queried for its mode.
***pinMode_P:** The pointer of the pin's mode
***isOutputDir_p:** The pointer of the pin's operation direction. If the pin is GPIO, it will return the direction of the GPIO otherwise it will return FL_GPIO_UNUSED.
For the pin to be operated in Gpio mode it has the following value:

```
typedef enum FlGpioDirectionTag
{
    FL_GPIO_UNUSED=0,
    FL_GPIO_INPUT = 1,
    FL_GPIO_OUTPUT
}FlGpioDirection;
```

Otherwise the value is **FL_GPIO_UNUSED**.

- **Return values**

FL_OK: Query of the pin mode is successful.
FL_RET_ERR_PARAM: Incorrect parameter
FL_RET_ERR_BAD_STATE: If the pin's status is unexpected

4.4.3 Periphery-SPI

Periphery-SPIs are the SPI bus service pins. These pins will be used in the following functions: For SIM900 and SIM900A, they are **FL_PIN_11**, **FL_PIN_12, FL_PIN_13, and FL_PIN_14**. Note that once these pins are configured as DISP, they cannot be configured as GPIO pins again. The maximal frequency of SPI clock is 13MHz and the minimal frequency is 50.78125KHz. It supports both 3-wire and 4-wire modems.

4.4.3.1 ebdat5_01SpiConfigure

The ebdat5_01SpiConfigure function subscribes to SPI bus service and sets eligible pins to be SPI pins: MISO, MOSI, SCLK and SS. To subscribe to SPI bus, these pins need to be unsubscribed from their default usage by this function first.

- **Prototype**

```
s32 ebdat5_01SpiConfigure(SsiModeType wireMode ,
                          SsiEnablePolarityType csPolHigh ,
                          FlPinName    cs_gpio_num,
                          SsiClockType clkSpeed ,
                          SsiDataPolarityType clkMode ,
                          SsiTrfFormatType msbFirst ) ;
```

- **Parameters**

SPI parameter is made up of following parameters.

wireMode:

SSI_3WIRE, for 3-wire mode SPI.

SSI_4WIRE, for 4-wire mode SPI.

For SIM900 and SIM900A:

3 Wire Mode				
SPI Name	**Platform Pin Name**	**Platform GPIOs**	**Direction**	**SIM900 and SIM900A Pin**
MOSI	SSI_DATA	GPIO50	output	DISP_DATA
SCLK	SSI_CLK	GPIO48	output	DISP_CLK
SS[2]	SSI_SEL1	GPIO52[1]	output	DISP_CS[1]
Notes:	***1. Platform GPIO52 is used.*** ***2. SS -- Slave Select.***			

4 Wire Mode				
SPI Name	**Platform Pin Name**	**Platform GPIOs**	**Direction**	**SIM900 and SIM900A Pin**
MISO	SSI_DATA	GPIO50	input	DISP_DATA
MOSI	SSI_OUT	GPIO49 / GPSR_CLK	output	DISP_D/C
SCLK	SSI_CLK	GPIO48	output	DISP_CLK
SS[2]	SSI_SEL1	GPIO52[1]	output	DISP_CS[1]
Notes:	***1. Platform GPIO52 is used.*** ***2. SS -- Slave Select.***			

csPolHigh:

SSI_ACTIVE_LOW, of low polarity

SSI_ACTIVE_HIGH, of high polarity

s_gpio_num:

gpio number used for SPI Chip Select

clkSpeed:

SSI_SYSTEM_DIV_2 /*26/2 Mhz*/

SSI_SYSTEM_DIV_4 /*26/4 Mhz*/

SSI_SYSTEM_DIV_8 /*26/8 Mhz*/

SSI_SYSTEM_DIV_16 /*26/16 Mhz*/
SSI_SYSTEM_DIV_32 /*26/32 Mhz*/
SSI_SYSTEM_DIV_64 /*26/64 Mhz*/
SSI_SYSTEM_DIV_128 /*26/128Mhz*/
SSI_SYSTEM_DIV_256 /*26/256Mhz*/
SSI_SYSTEM_DIV_512 /*26/512Mhz*/

clkMode :

SSI_FALLING_EDGE, write clock polarity is configured as falling edge
SSI_RISING_EDGE, write clock polarity is configured as rising edge

msbFirst:

SSI_LSBFIRST, to send LSB (least significant bit) data first
SSI_MSBFIRST, to send MSB (most significant bit) data first

- **Return values**

FL_OK: SPI Interface configuration is successful.
FL_ERROR: SPI Interface configuration is failed.

4.4.3.2 ebdat5_02SpiWriteByte

The ebdat5_02SpiWriteByte function writes one byte to the SPI interface.

- **Prototype**

```
s32 ebdat5_02SpiWriteByte(u8 data);
```

- **Parameters**

data: Byte to transfer

- **Return values**

FL_OK: Write byte successfully.
FL_ERROR: Write byte failed.

4.4.3.3 ebdat5_03SpiReadByte

The ebdat5_03SpiReadByte function will read one byte from the SPI interface.

- **Prototype**

```
u8 ebdat5_03SpiReadByte (void);
```

- **Parameters**

NONE

- **Return values**

One byte read from spi

4.4.3.4 ebdat5_04SpiWriteBytes

The ebdat5_04SpiWriteBytes function will write bytes to the SPI interface. This is a block function.

- **Prototype**

```
s32 ebdat5_04SpiWriteBytes(u8 *p_data, u32 dataSize);
```

- **Parameters**

p_data: Pointer of data to be sent.
dataSize: Size of data to be sent. It cannot exceed 4K bytes.

- **Return values**

FL_OK: Write bytes successfully.
FL_ERROR: Write bytes failed.

4.4.4 Periphery-Display

Periphery-Display is for displaying interface pins. These functions are used to control the screen of which its periphery bus is SPI. Following pins will be used in these function. For SIM900 and SIM900A, they are **FL_PIN_11**, **FL_PIN_12**, **FL_PIN_13**, **FL_PIN_14**.
Note that once these pins are configured as DISP, it cannot be configured as GPIO again. The maximal frequency of Display clock is 13MHz and the minimal frequency is 50.78125KHz.
Display interface is connected to SIM900 and SIM900A PINs: **FL_PIN_68** (used as **DISP_RST), DISP_D/C, DISP _DATA, DISP_CLK** and **DISP_CS**.

4.4.4.1 ebdat05_11DispConfig

The ebdat05_11DispConfig function configures display interface using SIM900 and SIM900A PINs: **FL_PIN_68** (used as **DISP_RST), DISP_D/C, DISP _DATA, DISP_CLK** and **DISP_CS**.

- **Prototype**

```
s32 ebdat05_11DispConfig ( FlPinName    cs_gpio_num,    SsiClockType clk);
```

- **Parameters**

cs_gpio_num: The GPIO used as Chip Select signal for display interface.
clk:

SSI_SYSTEM_DIV_2	**/***26/2 Mhz*/
SSI_SYSTEM_DIV_4	**/***26/4 Mhz*/
SSI_SYSTEM_DIV_8	**/***26/8 Mhz*/
SSI_SYSTEM_DIV_16	/*26/16 Mhz*/
SSI_SYSTEM_DIV_32	**/***26/32 Mhz*/
SSI_SYSTEM_DIV_64	**/***26/64 Mhz*/
SSI_SYSTEM_DIV_128	**/***26/128Mhz*/
SSI_SYSTEM_DIV_256	**/***26/256Mhz*/
SSI_SYSTEM_DIV_512	**/***26/512Mhz*/

- **Return values**

FL_OK: Display configuration successfully.
FL_ERROR: Display configuration failed.
Note: In order to use the SPI to display interface correctly, DO NOT configure these pins to a different mode before they are configured as DISP pins.

For SIM900 and SIM900A:

Display Interface				
SPI Name	**Platform Pin Name**	**Platform GPIOs**	**Direction**	**SIM900 and SIM900A Pin**
MOSI	SSI_DATA	GPIO50	output	DISP_DATA
SCLK	SSI_CLK	GPIO48	output	DISP_CLK
SS	SSI_SEL1	GPIO52	output	DISP_CS
--	GPIO1	GPIO1	output	GPIO12[1]
--	SSI_OUT	GPIO49 / GPSR_CLK	output	DISP_D/C
Note	***1. DISP_RST is SIM900 Pin GPIO12.***			

4.4.4.2 ebdat05_12DispWriteCommand

The ebdat05_12DispWriteCommand function sends one command (1 byte) to LED.
This operation will also clear **DISP_D/C** pin (low).

- **Prototype**

```
s32 ebdat05_12DispWriteCommand (u8 command);
```

- **Parameters**

command: The command to be sent to LED.

- **Return values**

FL_OK: Send display command successfully.
FL_ERROR: Send display command failed.

Note: In order to use the SPI to display interface correctly, DO NOT configure these pins to a different mode before they are configured as DISP pins.

4.4.4.3 ebdat05_13DispWriteData

The ebdat05_13DispWriteData function sends data (1 byte) to the display equipment.
This operation will also set **DISP_DC** pin (high).

- **Prototype**

```
s32 ebdat05_13DispWriteData (u8 data);
```

- **Parameters**

data: The data (1 byte) to be sent to the display equipment.

- **Return values**

FL_OK: Send display data successfully.
FL_ERROR: Send display data failed.

4.4.5 Periphery interrupt

Periphery interrupt functions can be used to configure the GPIO as GPIO interrupt.
The following is the description of the functions of SIM900 and SIM900A.
Note that only four pins can be used as GPIO interrupt. They are “FL_PIN_37”, “FL_PIN_38”, “FL_PIN_67” and “FL_PIN_68”.

4.4.5.1 ebdat6_13IntSubscribe

The ebdat6_13IntSubscribe function subscribes the pins to be interrupt pins, and changes the pin mode to be FL_PIN_FUNC_INTR. Please note that before the pin is configured as an interrupt, ebdat6_08pinConfigureToUnused must be called first to configure the pins to FL_PIN_MODE_UNUSED status. For eligible pins refer to section 4.4.5.

- **Prototype**

```
s32 ebdat6_13IntSubscribe(FlPinName pinName, FLGpioTriggerType triggerType,
                          u16    deBouncePeriodMs);
```

- **Parameters**

pinName: The pin which is configured as GPIO interrupt
triggerType:

```
typedef enum
{
    FL_GPIO_TRIG_ON_HIGH_LEVEL,     /*trigger on high level*/
    FL_GPIO_TRIG_ON_LOW_LEVEL,      /*trigger on low level*/
    FL_GPIO_TRIG_ON_RISING_EDGE,    /*trigger on rising edge*/
    FL_GPIO_TRIG_ON_FALLING_EDGE    /*trigger on rising edge*/
}FLGpioTriggerType;
```

deBouncePeriodMs: It is the debounce time of the interrupt. Its unit is millisecond. If it is less than 20ms, the debounce time will be ignored.

- **Return values**

FL_OK: Configure the pin to interrupt GPIO successfully.
FL_RET_ERR_BAD_STATE: If an error occurred.
FL_RET_ERR_PARAM: Incorrect parameter.

4.4.6 Periphery square wave

Periphery square wave interfaces are used to configure the PWM pin to generate PWM signal.

4.4.6.1 ebdat6_19SqWaveSubscribe

The ebdat6_19SqWaveSubscribe function assigns a square wave to generate PWM wave. There are two pins that user can use to generate PWM, which are PWM_1 and PWM_2.

- **Prototype**

```
s32 ebdat6_19SqWaveSubscribe(FlPWM pwm, u8 pwmhalfPeriod, u8 pwmlevel);
```

- **Parameters**

pwm: The PWM that user wants to generate.
pwmhalfPeriod: This is the period of the PWM. The period of PWM is equal to (pwmhalfPeriod + 1) / 3.25 MHz. Its range is from 0 to 126.
pwmlevel: This is the duty of PWM. It equals to the high level divided by the period of the PWM. Its range is from 0 to 100.

Note: pwmhalfPeriod is the frequency period; pwmlevel is the PWM pulse high time, which equals to high time / period.
eg:

ebdat6_19SqWaveSubscribe(FL_PWM_0, 100,50);
pwmhalfPeriod:100--->101 pwmclk
pwmlevel:50---->51 pwmclk
pwmclk=sysclk(26Mhz)/8=3.25Mhz
PWM out:3.25Mhz/101 = 32.178Khz
high time:51*pwmclk
In our reference code input level is limited.
if (level*period/100) = 0
then *pwmlevel* =127
if *pwmlevel* > pwmhalfPeriod
then pwm out low level
if user wants to set pwmclk=3.25Mhz/3=1.08Mhz
ebdat6_19SqWaveSubscribe(FL_PWM_0, 2,50);
pwmhalfPeriod:2--->3 pwmclk
50*2/100 = 1
pwmlevel:1---->2 pwmclk

- **Return values**

FL_OK: Subscribe the PWM successfully
FL_RET_ERR_PARAM: Incorrect parameter.

4.4.6.2 ebdat6_20SqWaveUnsubscribe

The ebdat6_20SqWaveUnsubscribe function unsubscribes **PWM** pin from square wave service, and changes the pin to low level.

- **Prototype**

```
s32 ebdat6_20SqWaveUnsubscribe(FlPWM pwm) ;
```

- **Parameters**

pwm: The PWM that user wants to generate.

- **Return values**

FL_OK: Unsubscribe the PWM successfully.
FL_RET_ERR_PARAM: Incorrect parameter.

4.4.7 Periphery-GPIO

Periphery GPIO interfaces are used to configure pins to be GPIO. It can also be used to set the GPO's level and read the level from the GPI. Note that FL_PIN_3 cannot be configured as GPIO, as it is reserved.

4.4.7.1 ebdat6_02GpioSubscribe

The **ebdat6_02GpioSubscribe** function subscribes pins to GPIO pins and changes pin mode to **FL_PIN_MODE_GPIO**. Before this function is called, ebdat6_08pinConfigureToUnused should be called to configure the pin mode to **FL_PIN_MODE_UNUSED** status.

- **Prototype**

```
s32 ebdat6_02GpioSubscribe(FlPinName pinName,
                           FlGpioDirection gpioDir,
                           bool defValue);
```

- **Parameters**

pinName: Refer to Appendix A for eligible pins. Note that FL_PIN_3 cannot be configured as GPIO, as it is reserved.
gpioDir: Input/output direction of the pin, refer to the pin lists for details on eligible pins. Some pins can only be assigned as input while others can only be assigned as output pins.
defValue: Gpo default value.

- **Return values**

FL_OK: Subscribe the pin to GPIO successfully.
FL_RET_ERR_BAD_STATE: If an error occurred.
FL_RET_ERR_PARAM: Incorrect parameter.

4.4.7.2 ebdat6_05ReadGpio

The ebdat6_05ReadGpio function reads the level from GPI pins. The pin should be configured as GPI first.

- **Prototype**

```
s32 ebdat6_05ReadGpio(FlPinName pinName, bool *inputValue_p);
```

- **Parameters**

pinName: The name of the GPIO pin from which the level to be read. Note that FL_PIN_3 cannot be configured as GPIO, as it is reserved.
***inputValue_p:** Pointer to the read value

- **Return values**

FL_OK: On success.
FL_RET_ERR_BAD_STATE: If an error occurs. Check whether the pin has been configured as GPI or not.

FL_RET_ERR_PARAM: Incorrect parameter.

4.4.7.3 ebdat6_04WriteGpio

The ebdat6_04WriteGpio function writes to GPIO pins. The pin should be configured as GPO first.

- **Prototype**

```
s32 ebdat6_04WriteGpio(FlPinName pinName, bool outputValue);
```

- **Parameters**

pinName: The name of the GPIO pin to which the level to be written. Note that FL_PIN_3 cannot be configured as GPIO, as it is reserved.
outputValue: The value to be written to the pin

- **Return values**

FL_OK: Set the GPO to expected level successfully.
FL_RET_ERR_BAD_STATE: If an error occurred. Please check if the pin has been configured as GPO.
FL_RET_ERR_PARAM: Incorrect parameter.

4.4.8 Periphery-Keypad

Periphery-Keypad interfaces are used to configure pins to be keypad. Only following pins can be used as key pad pins. They are FL_PIN_40, FL_PIN_41, FL_PIN_42, FL_PIN_43, FL_PIN_44, FL_PIN_47, FL_PIN_48, FL_PIN_49, FL_PIN_50, FL_PIN_51. Note that once one of these pins is configured as GPIO, the rest of them will all be configured to GPI automatically.

4.4.8.1 ebdat6_15KeySubscribe

The ebdat6_15KeySubscribe function initializes the keypad pins to be keypad.

- **Prototype**

```
s32 ebdat6_15KeySubscribe(void);
```

- **Return values**

FL_OK: Initialize successfully.

4.5 Audio API

File fl_audio.h needs to be included before audio functions are called.

4.5.1 ebdat10_01PlayContinousAudio

The ebdat10_01PlayContinousAudio function plays the continuous music in system.

- **Prototype**

```
bool ebdat10_01PlayContinousAudio(FlAudioName name);
```

- **Parameters**

name: The audio track name and its range must be from FL_MELODY01 to FL_DIAL_TONE.

- **Return values**

TRUE: If it is ok, otherwise it will return FAIL.

4.5.2 ebdat10_02StopContinousAudio

The ebdat10_02StopContinousAudio function stops playing continuous music

- **Prototype**

```
bool ebdat10_02StopContinousAudio(void) ;
```

- **Return values**

TRUE: If it is ok, if not it will return FAIL.

4.5.3 ebdat10_03PlaySingleAudio

The ebdat10_03PlaySingleAudio function plays the audio one time. Its range must be from FL_SUBSCRIBER_BUSY_TONE to FL_GAME_OVER.

- **Prototype**

```
bool ebdat10_03PlaySingleAudio(FlAudioName name) ;
```

- **Parameters**

name: The audio track name and its range must be from **FL_SUBSCRIBER_BUSY_TONE** to

FL_GAME_OVER.

- **Return values**

TRUE: If it is ok
FALSE: If it is failed.

4.5.4 ebdat10_04PlaySingleAudioFromFile

The ebdat10_04PlaySingleAudioFromFile function is used to play an audio file which is stored in the flash.

- **Prototype**

```
bool ebdat10_04PlaySingleAudioFromFile(u8* fileName);
```

- **Parameters**

fileName: The audio file name which is to be played.

- **Return values**

TRUE: If it is ok
FALSE: If it is failed.

4.5.5 AUDIO TRACKS

```
typedef enum FlAudioNameTag
{
   /*Continous*/
   FL_MELODY01 = 0,
   FL_MELODY02,
   FL_MELODY03,
   FL_MELODY04,
   FL_MELODY05,
   FL_MELODY06,
   FL_MELODY07,
   FL_MELODY08,
   FL_MELODY09,
   FL_MELODY10,
   FL_MELODY11,
   FL_MELODY12,
   FL_MELODY13,
   FL_MELODY14,
```

```
    FL_MELODY15,
    FL_MELODY16,
    FL_MELODY17,
    FL_MELODY18,
    FL_MELODY19,
    FL_MELODY20,
    FL_CALL_WAITING,
    FL_RINGING_TONE,
    FL_DIAL_TONE,
    /*Single*/
    FL_SUBSCRIBER_BUSY_TONE,
    FL_CONGESTION,
    FL_RADIO_PATH_NOT_AVAILABLE,
    FL_RADIO_PATH_ACKNOWLEDGED,
    FL_NUMBER_UNOBTAINABLE,
    FL_POSITIVE_SOUND_KISS,
    FL_NEGATIVE_SOUND_KISS,
    FL_ERROR_BEEP_KISS,
    FL_SWITCH_ON,
    FL_SWITCH_OFF,
    FL_BUMPER_SOUND,
    FL_KEY_TONE,
    FL_NEW_OCCURENCE_SOUND,
    FL_ALARM_SOUND,
    FL_AUTOREDIALSTART,
    FL_AUTOREDIALSUCCES,
    FL_GAME_INTRO,
    FL_GAME_NEW_LEVEL,
    FL_GAME_NEW_HIGH_SCORE,
    FL_GAME_LOSE_LIFE,
    FL_GAME_OVER,
    FL_AUDIO_INVALID }
FlAudioName;
```

4.6 TIMER API

File fl_timer.h needs to be included for the following APIs to work properly. In this part, the interfaces are used to start or stop a timer or get the system tick or time. Note that only 10 timers can be started at the same time.

4.6.1 Timer structure

```
typedef struct FlTimerTag
{
    u32         timeoutPeriod; /*the time elapse before the timer expires*/
    u16         timerId; /* the ID of the timer*/
}
t_emb_Timer;
```

4.6.2 ebdat8_01StartTimer

The ebdat8_01StartTimer function starts a timer. When the timer is expired, it will be stopped and if another time period is wanted, the "ebdat8_01StartTimer" must be called to start the timer again.

- **Prototype**

```
s32 ebdat8_01StartTimer(t_emb_Timer timer);
```

- **Parameters**

timer: The timer to be started. This variable has two members. The timeoutPeriod is the time elapsed before the timer expires. The timerId is the ID of the timer.

- **Return values**

FL_RET_ERR_PARAM: Incorrect parameter.
FL_RET_ERR_BAD_STATE: The timer has been started.
FL_OK: Start a timer successfully.

Example:

```
t_emb_Timer timerDemo;
timerDemo.timeoutPeriod = ebdat8_04SecondToTicks(2); /* set timeout to be 2 seconds*/
if (ebdat8_01StartTimer(timerDemo) == FL_OK)
{
    ….
```

```
} /*start the timer*/
/* for time out event, refer to 3.1.6 EVENT_TIMER section*/
```

4.6.3 ebdat8_02StopTimer

The ebdat8_02StopTimer function stops a Timer before it expires.

- **Prototype**

```
u16 ebdat8_02StopTimer(t_emb_Timer timer);
```

- **Parameters**

timer: The timer to be stopped. This variable has two members. The timeoutPeriod is the time elapsed before the timer expires. The timerId is the ID of the timer.

- **Return values:**

FL_RET_ERR_PARAM: Incorrect parameter.
FL_OK: Stop a timer successfully.

4.6.4 ebdat8_04SecondToTicks

The ebdat8_04SecondToTicks function converts time from seconds to KernelTicks.
One kernel tick = 9.23 milliseconds.

- **Prototype**

```
u32 ebdat8_04SecondToTicks(u32 seconds);
```

- **Parameters**

seconds: It is the time expected to be converted. Its unit is second.

- **Return values**

The return value is measured in KernelTicks.

4.6.5 ebdat8_05MillisecondToTicks

The ebdat8_05MillisecondToTicks function converts time from milliseconds to KernelTicks.

- **Prototype**

```
u32 ebdat8_05MillisecondToTicks(u32 milliseconds);
```

- **Parameters**

milliseconds: It is the time that is expected to be converted. Its unit is millisecond.

- **Return values**

The return value is measured in KernelTicks.

4.6.6 ebdat8_03GetRelativeTime

The ebdat8_03GetRelativeTime function gets the rest of ticks before the timer will be expired.

- **Prototype**

```
s32 ebdat8_03GetRelativeTime(t_emb_Timer timer, u32 *tick) ;
```

- **Parameters**

timer: The timer to be stopped. This variable has two members. The timeoutPeriod is the time elapsed before the timer expires. The timerId is the ID of the timer.
***tick:** It will return the rest of ticks that the timer will be expired.

- **Return values**

FL_OK: Get the system time successfully
FL_RET_ERR_PARAM: Incorrect parameter

4.6.7 ebdat8_06GetSystemTime

The ebdat8_06GetSystemTime function gets the local time.

- **Prototype**

```
void ebdat8_06GetSystemTime(t_emb_SysTimer * datetime);
```

- **Parameters**

datetime: An t_emb_SysTimer struct to store current local time.
t_emb_SysTimer are defined as:

```
typedef struct FlSysTimerTag
{
    unsigned short year;
```

```
    unsigned char month;
    unsigned char day;
    unsigned char hour;
    unsigned char minute;
    unsigned char second;
}t_emb_SysTimer;
```

4.6.8 ebdat8_08GetSystemTickCounter

The ebdat8_08GetSystemTickCounter function gets the system ticks when the module is powered on.

- **Prototype**

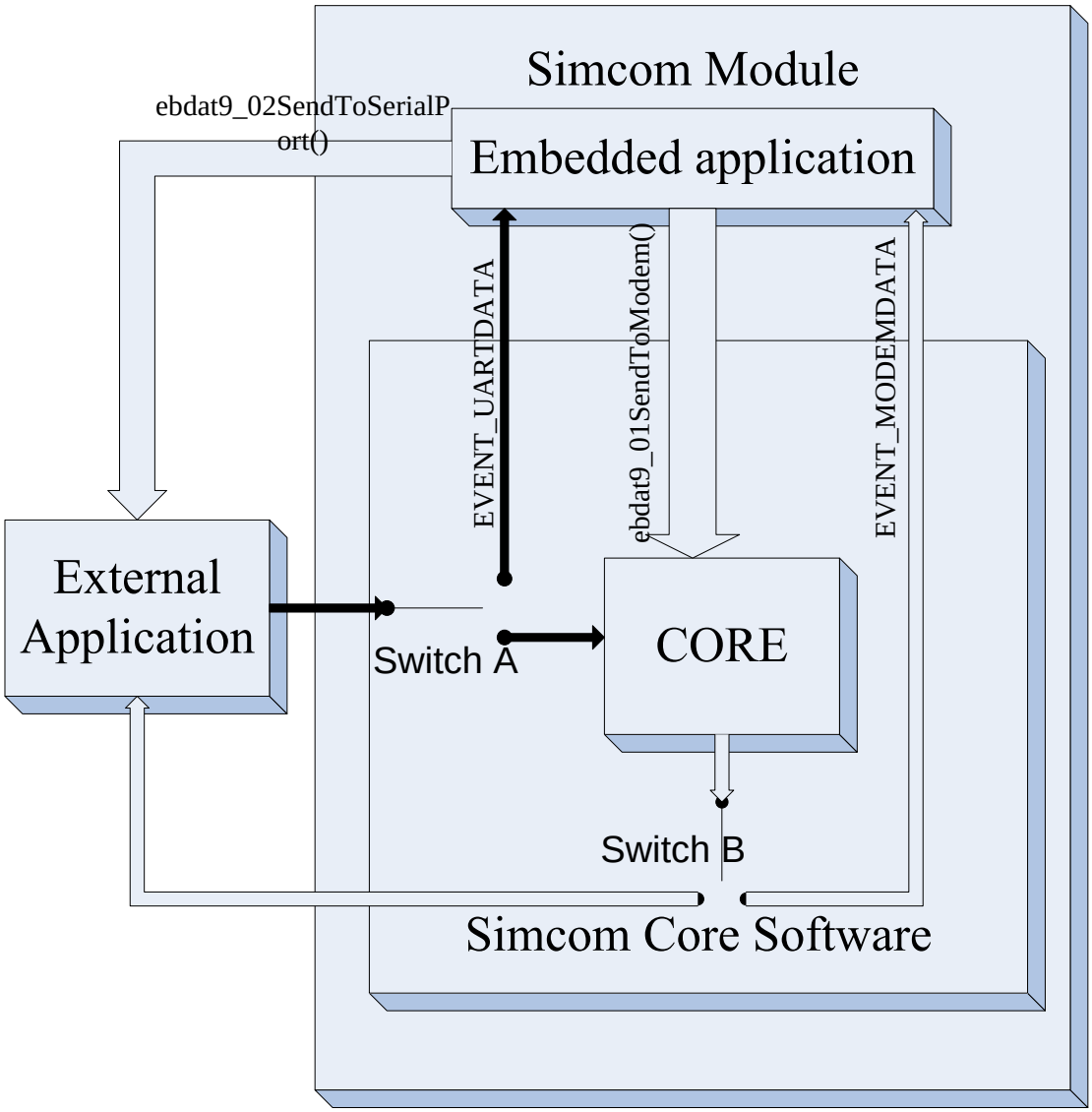
```
u32 ebdat8_08GetSystemTickCounter(void);
```

- **Return values**

It returns the system ticks when the module is powered on.

4.7 FCM API

File fl_fcm.h needs to be included for these APIs to work.
The following diagram illustrates how each FCM function controls the direction of data flow



Switch A: ebdat9_04SetUartdataToFL function
Switch B: ebdat9_03SetModemdataToFL function
CORE: Core data flow control software.

Switch A is the input flow director, when it is set to 1, data coming from external application (trace port or serial port) will be directed to the embedded application, and triggers EVENT_UARTDATA event. When it is set to 0, external data will flow into SIMCom core software, and it no longer notifies embedded application.

Switch B is the output flow director, when it is set to 1, data coming out of SIMCom core software

will go to the embedded application, and trigger EVENT_MODEMDATA. When it is 0, data will go directly to the external application, and no data is received by embedded application.

4.7.1 ebdat9_01SendToModem

This function sends data to core buffer. Return information of AT commands and result codes OK or ERROR are received by eat1_02GetEvent function when ebdat9_03SetModemdataToFL is set to 1. Refer to Chapter 3.1.5 for more details. A special character "\r" (cartridge return) should be appended to the string of AT command to indicate the end of it. For example: **ebdat9_01SendToModem ("ati\r",4)** is same as user typing "ati" command and pressing ENTER.

- **Prototype**

```
s32 ebdat9_01SendToModem(u8 *senddata,u16 data_len);
```

- **Parameters**

senddata: The data which will go into core buffer.
data_len: The length of the data, which cannot exceed 1024.

- **Return values**

FL_OK: Send to modem successfully.
FL_RET_ERR_PARAM: Incorrect parameter

4.7.2 ebdat9_02SendToSerialPort

The ebdat9_02SendToSerialPort function is used to send string to serial port, it is valid only when ebdat9_05GetSerialPortTxStatus returns 1 (which means the transmit buffer is null).

- **Prototype**

```
s32 ebdat9_02SendToSerialPort(char *src, u16 len);
```

- **Parameters**

src: The string user wants to send to serial port.
len: The length of the string, which must be less than 256.

- **Return values**

FL_OK: Send to serial port successfully.
FL_RET_ERR_PARAM: Incorrect parameter.

4.7.3 ebdat9_03SetModemdataToFL

The ebdat9_03SetModemdataToFL function controls output data's direction from core.

- **Prototype**

```
void ebdat9_03SetModemdataToFL (bool destination);
```

- **Parameters**

destination:

TRUE: Sends the output data from core to embedded application.
FALSE: It is directed to serial port.

4.7.4 The ebdat9_04SetUartdataToFL function

The ebdat9_04SetUartdataToFL function controls the input data's direction from serial port.

- **Prototype**

```
void ebdat9_04SetUartdataToFL (bool destination);
```

- **Parameters**

destination:

TRUE: The input data from serial port is sent to embedded application.
FALSE: For sending to core buffer.

4.7.5 ebdat9_05GetSerialPortTxStatus

The ebdat9_05GetSerialPortTxStatus function gets the transmit buffer's status of the serial port. If it returns FALSE, user cannot send any data to serial port.

- **Prototype**

```
bool ebdat9_05GetSerialPortTxStatus(void);
```

- **Return values**

TRUE: The transmit buffer is null, data can be sent to the serial port.
FALSE: There are data in the transmit buffer.

4.7.6 ebdat6_23GetRTSPinLevel

The ebdat6_23GetRTSPinLevel function is used to get the status of RTS level. If it returns 1, it means that RTS is high level. Otherwise it means that RTS is low level.

- **Prototype**

```
u8 ebdat6_23GetRTSPinLevel (void);
```

- **Return values**

1: RTS is high level.
0: RTS is low level.

4.7.7 ebdat9_09ChangeMainUartBaudRate

The ebdat9_09ChangeMainUartBaudRate function sets the baud rate of the main serial port.

- **Prototype**

```
s32 ebdat9_09ChangeMainUartBaudRate(u32 BaudRate);
```

- **Parameters**

BaudRate: The baud rate of the main port. The range of its value is 0, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200. Note that ‘0’ means auto baud.

- **Return values**

FL_OK: Set the baud rate successfully.
FL_ERROR: Set baud rate failed.

4.7.8 ebdat9_10GetMainUartBaudRate

The ebdat9_10GetMainUartBaudRate function is used to get the baud rate of the main serial port.

- **Prototype**

```
u32 ebdat9_10GetMainUartBaudRate(void);
```

- **Return values**

It returns the baud rate of the main serial port. Its range is 0, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200. Note that ‘0’ means auto baud.

4.7.9 ebdat9_11ChangeMainUartDataFormat

The ebdat9_11ChangeMainUartDataFormat function sets the data format of the main port.

- **Prototype**

```
s32 ebdat9_11ChangeMainUartDataFormat(FlMainUartDataFormat uartDataFormat);
```

- **Parameters**

uartDataFormat:

```
typedef enum FlMainUartFormatTag
{
    FL_MAIN_UART_8N2_FORMAT = 1,   /*8 data 0 parity 2 stop*/
    FL_MAIN_UART_8P1_FORMAT = 2,   /*8 data 1 parity 1 stop*/
    FL_MAIN_UART_8N1_FORMAT = 3,   /*8 data 0 parity 1 stop*/
    FL_MAIN_UART_7N2_FORMAT = 4,   /*7 data 0 parity 2 stop*/
    FL_MAIN_UART_7P1_FORMAT = 5,   /*7 data 1 parity 1 stop*/
    FL_MAIN_UART_7N1_FORMAT = 6    /*7 data 0 parity 1 stop*/
}FlMainUartFormat;
```

```
typedef enum FlMainUartParityTag
{
    FL_MAIN_UART_ODD = 0,          /*odd parity*/
    FL_MAIN_UART_EVEN = 1,         /*even parity*/
    FL_MAIN_UART_SPACE = 3         /*space parity*/
}FlMainUartParity;
```

```
typedef struct FlMainUartDataFormatTag
{
    FlMainUartFormat uartFormat;
    FlMainUartParity uartParity;
}FlMainUartDataFormat;
```

- **Return values**

FL_OK: Set the data format successfully.
FL_ERROR: Set data format failed.

4.7.10 ebdat9_12GetMainUartDataFormat

The ebdat9_12GetMainUartDataFormat function is used to get the data format of the main port.

- **Prototype**

```
FlMainUartDataFormat ebdat9_12GetMainUartDataFormat(void);
```

- **Return values**

Return the data format of the main port. The structure is defined in 4.7.9.

4.7.11 ebdat9_13ChangeMainUartFlowControl

The ebdat9_13ChangeMainUartFlowControl function sets the flow control of the main port.

- **Prototype**

```
s32 ebdat9_13ChangeMainUartFlowControl(FlMainUartFlowControlStruct flowControl);
```

- **Parameters**

flowControl:

```
typedef enum FlMainUartFlowControlTag
{
    FL_MAIN_UART_NO_FLOW_CONTROL,          /*No flow control.*/
    FL_MAIN_UART_SOFTWARE_FLOW_CONTROL,    /*software flow control*/
    FL_MAIN_UART_HARDWARE_FLOW_CONTROL     /*hardware flow control*/
}FlMainUartFlowControl;
```

```
typedef struct FlMainUartFlowControlStructTag
{
    FlMainUartFlowControl dcebydte;     /*To specifies the method will be used by TE at
                                          receive of data from TA*/
    FlMainUartFlowControl dtebydce;
}FlMainUartFlowControlStruct;
```

- **Return values**

FL_OK: Set the data format successfully.
FL_ERROR: Set data format failed.

4.7.12 ebdat9_14GetMainUartFlowControl

The ebdat9_14GetMainUartFlowControl function is used to get the flow control of the main port.

- **Prototype**

```
FlMainUartFlowControlStruct ebdat9_14GetMainUartFlowControl(void);
```

- **Return values**

Return the flow control of the main port. The structure is defined in 4.7.11.

4.7.13 ebdat9_15SubscribeURC

The ebdat9_15SubscribeURC function subscribes a URC. When modem sends to the URC which is subscribed, a call-back function will be called.

- **Prototype**

```
s32 ebdat9_15SubscribeURC(u8 *urcString, u32 stringLen, fl_urchandle hd,
                          u8 isWholeStringCmp);
```

- **Parameters**

urcString: The URC to be subscribed. The maximum of the URC which can be subscribed is **32**.
stringLen: The length of the URC string.
hd: The call back function

```
typedef void(*fl_urchandle)(u8 *data, u32 datalen);
```

isWholeStringCmp: if it is 1, the URC should be identical to the string which was set, and the call back function will be called. If it is 0, the string which was set is one part of the URC, and the call back function will be called.

- **Return values**

FL_RET_ERR_PARAM: Incorrect parameter
FL_RET_ERR_ALREADY_SUBSCRIBED: The URC has been subscribed.
FL_ERROR: Subscribe the URC failed. The number of URC reaches the maximum number.
FL_OK: Subscribe the URC successfully.

4.7.14 ebdat9_16UnSubscribeURC

The ebdat9_16UnSubscribeURC function is used to unsubscribe a URC.

- **Prototype**

s32 ebdat9_16UnSubscribeURC(u8 *urcString);

- **Parameters**

urcString: The URC will be unsubscribed.

- **Return values**

FL_RET_ERR_PARAM: Incorrect parameter
FL_RET_ERR_NOT_SUBSCRIBED: The URC has not been subscribed.
FL_ERROR: Subscribe the URC fail.
FL_OK: Unsubscribe the URC successfully.

4.7.15 ebdat9_17GetURCNum

The ebdat9_17GetURCNum function is used to get the number of URCs which have been subscribed.

- **Prototype**

u8 ebdat9_17GetURCNum(void);

- **Return values**

Return the number of URCs which have been subscribed.

4.7.16 ebdat9_19SubscribeATCommand

The ebdat9_19SubscribeATCommand function subscribes an AT command which can be defined by the customer. The maximum number of the AT command is **32**.

- **Prototype**

s32 ebdat9_19SubscribeATCommand(ascii *urcString, u32 index);

- **Parameters**

urcString: The AT command will be unsubscribed. The maximum of the AT command which can be subscribed is 8. And its length cannot exceed 20 bytes.
index: index which corresponded to the AT command. It cannot be 0xFFFFFFFF.

EVENT_MODEM event will be triggered when module receives AT command customized by customer. Customer can use the variable atCommandIndex of MODEMDATA_EVT structure to acquire which AT command is triggered (atCommandIndex is correlated with index)

- **Return values**

FL_RET_ERR_PARAM: Incorrect parameter
FL_RET_ERR_ALREADY_SUBSCRIBED: The AT command has been subscribed.
FL_ERROR: Subscribe the URC fail. The number of AT commands reaches the maximum number.
FL_OK: Subscribe the URC successfully.

4.7.17 ebdat9_20UnsubscribeATCommand

The ebdat9_20UnsubscribeATCommand function unsubscribes an AT command which can be defined by the customer.

- **Prototype**

```
s32 ebdat9_20UnsubscribeATCommand(ascii *pString);
```

- **Parameters**

urcString: The AT command will be unsubscribed.

- **Return values**

FL_RET_ERR_PARAM: parameter error
FL_RET_ERR_ALREADY_SUBSCRIBED: The URC has been subscribed.
FL_ERROR: Subscribe the URC failed. The number of URC reaches the maximum number.
FL_OK: Subscribe the URC successfully.

4.8 Debug API

File fl_trace.h must be included for debug functions to work.

4.8.1 ebdat7_00EnterDebugMode

The ebdat7_00EnterDebugMode function enters debug mode, once in debug mode, ebdat7_01DebugTrace() prints debug information to the debug port instead of sending debug information to spytrace. The default debug mode is off.

- **Prototype**

```
void ebdat7_00EnterDebugMode(void);
```

4.8.2 ebdat7_01DebugTrace

The ebdat7_01DebugTrace function prints out customer's data to debug port.

- **Prototype**

```
void ebdat7_01DebugTrace (const u8 *Format, ... );
```

- **Parameters**

Format: The parameter string works identical to printf function, except for:

"**\r**" Outputs to the beginning of a line, equivalent of "**\x0d**".

"**\n**" Outputs to a new line, but vertical prompt position remains the same from its last position, equivalent of "**\x0a**".

Note: In order to print from the beginning of a new line, the combination of "\r\n" will be used.

4.8.3 ebdat7_02DebugUartSend

The ebdat7_02DebugUartSend function prints out customer data to debug port. This is a block function.

- **Prototype**

```
s32 ebdat7_02DebugUartSend(u8 *buff, u16 len);
```

- **Parameters**

buff: The data user wants to send to the trace port.
len: The length of data user wants to send to the trace port.

- **Return values**

FL_ERROR: If the len is larger than 512 or buff, it is NULL.
FL_OK: Send the data successfully.

4.9 Standard library API

STDLIB API includes standard library function definitions in the file "fl_stdlib.h"

4.9.1 Standard input/output functions

```
#define fl_strcpy strcpy
#define fl_strncpy strncpy
```

```
#define   fl_strcat          strcat
#define   fl_strncat         strncat
#define   fl_strlen          strlen
#define   fl_strcmp          strcmp
#define   fl_strncmp         strncmp
#define   fl_strnicmp        strnicmp
#define   fl_memset          memset
#define   fl_memcpy          memcpy
#define   fl_memcmp          memcmp
#define   fl_itoa            itoa
#define   fl_atoi            atoi
#define   fl_sprintf         sprintf
#define   fl_memmove         memmove
```

Note: Above STDIO functions are identical to their standard C counter parts, the only difference is that these functions use user defined types instead of standard C types.

4.9.2 ebdat4_10strRemoveCRLF

The ebdat4_10strRemoveCRLF function removes the cartridge return "/r" and line feeder "/n" character from a string

- **Prototype**

```
ascii * ebdat4_10strRemoveCRLF ( ascii * dst, ascii * src, u16 size );
```

- **Parameters**

***dst:** Modified string
***src:** Original string
size: Size of the original string

- **Return values**

Modified string

4.9.3 ebdat4_11strGetParameterString

The ebdat4_11strGetParameterString function returns parameter string at a given position

- **Prototype**

```
ascii * ebdat4_11strGetParameterString ( ascii * dst, const ascii * src, u8 Position );
```

- **Parameters**

***dst:** Destination string
***src:** Original string
Position: Parameter position

- **Return values**

Address to the parameter string

4.9.4 ebdat6_17DisablePowerOffKey

The ebdat6_17DisablePowerOffKey function makes power key as a normal key instead of a power off key.

- **Prototype**

```
void    ebdat6_17DisablePowerOffKey ( void );
```

4.9.5 ebdat6_18EnablePowerOffKey

The ebdat6_17EnablePowerOffKey function makes power key as a power key instead of a normal key.

- **Prototype**

```
void    ebdat6_18EnablePowerOffKey ( void );
```

4.9.6 ebdat4_15ExitOutOfSleepMode

The ebdat4_15ExitOutOfSleepMode function makes the module go out of sleep mode.

- **Prototype**

```
s32 ebdat4_15ExitOutOfSleepMode(void);
```

- **Return values**

FL_OK: exit sleep mode successfully.
FL_ERROR: exit sleep mode failed.

4.9.7 ebdat4_17EnterSleepMode

The ebdat4_17EnterSleepMode function makes the module go into sleep mode.
Note: Before calling this function, "AT+CSCLK=2" should be sent to the Modem first.

- **Prototype**

```
s32 ebdat4_17EnterSleepMode(void);
```

- **Return values**

FL_OK: enter sleep mode successfully.
FL_ERROR: enter sleep mode failed.

4.10 SOCKET API

SOCKET APIs are used for TCP/IP data operation with API forms in the Embedded AT program. API method is designed to satisfy the customers who used to use API, customers still can use AT command in Embedded AT of SIM900 to get more powerful APPTCP, FTP, HTTP and TCP/IP data operation.

4.10.1 ebdat11_10GprsActive

The ebdat11_10GprsActive function is used to activate gprs bearer.

- **Prototype**

```
s32 ebdat11_10GprsActive(u8 *apnName,u8 *user,u8 *pass);
```

- **Parameters**

***apnName:** The APN of the bearer to be activated, which is 32 bytes long maximum
***user：** The user name of the bearer to be activated, which is 32 bytes long maximum
***pass：** The password of the bearer to be activated, which is 32 bytes long maximum

- **Return values**

FL_OK: Legal parameter, start to activate gprs scenario.
FL_ERROR: Illegal parameter or gprs was already activated.

- **Related EVENT**

The result of GPRS activation, it will be returned through **EVENT_SOCKET** among which type is **FL_SOCKET_GPRS_ACTIVE**, bsdResult 0 means activation failure, 1 means activation successful.

4.10.2 ebdat11_15GprsDeactive

The ebdat11_15GprsDeactive function is used to release gprs bearer.

- **Prototype**

```
s32 ebdat11_15GprsDeactive(void);
```

- **Return values**

FL_OK: Legal parameter, start to release gprs scenario.
FL_ERROR: gprs scenario was not activated and cannot be released.

- **Related EVENT**

GPRS activation result, it will be returned through **EVENT_SOCKET**, among which type is **FL_SOCKET_GPRS_DEACTIVE**, bsdResult 0 means gprs release failure, 1 means release successful.

Note: If network initiates the release of GPRS scenario, it is also reported through EVENT_SOCKET, among which type is FL_SOCKET_GPRS_DEACTIVE, bsdResult is 1.

4.10.3 ebdat11_20SocketConnect

The ebdat11_20SocketConnect function sets up TCP andUDP socket.

- **Prototype**

```
u32 ebdat11_20SocketConnect(FlSocketType_e type,u8 *url, u16 sockPort);
```

- **Parameters**

*** type:** EBDAT_TCP_CONNECT represents TCP, EBDAT_UDP_CONNECT represents UDP.
*** url:** The remote IP or domain name of the socket
sockPort: The remote port number of the socket

- **Return values**

Socket id, used for closing, sending and receiving data operation. If it is 0XFFFFFFF, it means setup failed.

- **Related EVENT**

The result of connect, it will be returned through **EVENT_SOCKET**, among which type is **FL_SOCKET_CONNECT**, socket id is the return value of ebdat11_20SocketConnect, bsdResult 0 means socket close failure, 1 means close successful.

4.10.4 ebdat11_25SocketClose

The ebdat11_25SocketClose function is used to close the socket.

- **Prototype**

```
s32 ebdat11_25SocketClose(u32 socket,u8 mode);
```

- **Parameters**

Socket: Socket id for those to be closed
mode: 0 Close by FIN method
1 Close by RST method

- **Return values**

FL_OK: Legal parameter, start to close the socket.
FL_ERROR: Socket has not been set up, and can not be closed.

- **Related EVENT**

The result of close, it will be returned through **EVENT_SOCKET**, among which type is **FL_SOCKET_CLOSE**, socket id is the return value of ebdat11_20SocketConnect, bsdResult 0 means socket close failure, 1 means close successful.

Note：If the connection is closed remotely, the result will be returned through EVENT_SOCKET, among which type is FL_SOCKET_REMOTE_CLOSE, socket id is the return value of ebdat11_20SocketConnect.

4.10.5 ebdat11_30SocketSend

The ebdat11_30SocketSend function sends socket data.

- **Prototype**

```
s32 ebdat11_30SocketSend(u32 socket,void *buf_p, u16 len);
```

- **Parameters**

Socket**:** The socket id for those data to be sent
***buf_p:** The data pointer to be sent
len: The data length to be sent

- **Return values**

FL_OK: Legal parameter, start to send data.
FL_ERROR: Parameter error or status error

- **Related EVENT**

The result of Send, it is returned through **EVENT_SOCKET**, among which type is **FL_SOCKET_SEND**, socket id is the return value of ebdat11_20SocketConnect. If bsdResult is 0 it means send failed, other value represents the length of data received by protocol stack.

Note: It will be used here only when module needs to wait for the return of FL_SOCKET_SEND event after ebdat11_30SocketSend. Generally the return value of bsdResult in FL_SOCKET_SEND event equals the len parameter of ebdat11_30SocketSend, if it does not equal or is 0, it means abnormal data sent, user needs to wait for some time then retry to send data.

4.10.6 ebdat11_35SocketRecv

The ebdat11_35SocketRecv function is used to read socket data.

- **Prototype**

u16 ebdat11_35SocketRecv(u32 socket,void *buf_p, u16 len,u16 *remain);

- **Parameters**

socket: Socket id of data to be read
***buf_p:** buffer of data to be read
len: max data length to be read
***remain:** The data length which can be acquired by this return value when the function is called, this data length is not an accurate value, the actual data length may be much greater than ***remain.**

- **Return values**

The data length when read is successful.

- **Related EVENT**

After receiving **EVENT_SOCKET**, among which type is **FL_SOCKET_RECV**, socket id is the return value of ebdat11_20SocketConnect, bsdResult is readable data length. The data can be acquired by ebdat11_35SocketRecv.

4.10.7 ebdat11_45SocketTcpServerSet

Set up and close tcp server.

- **Prototype**

```
S32 ebdat11_45SocketTcpServerSet(u8 mode,u16 port);
```

- **Parameters**

mode: 1 means socket setup successfully, 0 means closing server.
port: Local monitor port, this parameter will not be examined when mode is 0.

- **Return values**

FL_OK: Legal parameter, which is operating.
FL_ERROR: Parameter error or status error

- **Related EVENT**

EVENT_SOCKET will be received when Server is successfully setup, among which event type is **FL_SOCKET_TCP_SERVER_START**, socket id is the socket id of the server, if bsdResult is 0, it means server setup failed, while 1 means setup is successful.

EVENT_SOCKET will be received when Server is successfully closed, among which Event type is **FL_SOCKET_TCP_SERVER_STOP**, socket id is the socket id of the server, if bsdResult is 0, it means server close failed, while 1 means close is successful.

EVENT_SOCKET will be received when client is connected to server, among which Event type is **FL_SOCKET_TCP_SERVER_CONNECT**, socket id is the socket id assigned to this connection, which equals the return value of ebdat11_20SocketConnect. It can be used for closing, sending and receiving data operation.

4.11 Error Codes

fl_error.h defines all the error codes API function may return.

Error code	Error value	Description
FL_OK	0	No error response
FL_ERROR	-1	General error code
FL_RET_ERR_PARAM	-2	Parameter error
FL_RET_ERR_UNKNOWN_HDL	-3	Unknown handler / handle error
FL_RET_ERR_ALREADY_SUBSCRIBED	-4	Service already subscribed
FL_RET_ERR_NOT_SUBSCRIBED	-5	Service not subscribed
FL_RET_ERR_FATAL	-6	Fatal error
FL_RET_ERR_BAD_HDL	-7	Bad handle
FL_RET_ERR_BAD_STATE	-8	Bad state
FL_RET_ERR_PIN_KO	-9	Bad PIN state
FL_RET_ERR_NO_MORE_HANDLES	-10	The maximum service subscription capacity is reached
FL_RET_ERR_SPECIFIC_BASE	-20	Beginning of specific error range
FL_RET_ERR_OVERSIZE	-11	The Embedded application Update file is too big
FL_RET_ERR_UNMATCH	-12	The embedded application update file size does not match the function parameter

Flash related error code

Error code	Error value
FL_FLH_RET_ERR_OBJ_NOT_EXIST	FL_RET_ERR_SPECIFIC_BASE
FL_FLH_RET_ERR_MEM_FULL	FL_RET_ERR_SPECIFIC_BASE-1
FL_FLH_RET_ERR_NO_ENOUGH_IDS	FL_RET_ERR_SPECIFIC_BASE-2
FL_FLH_RET_ERR_ID_OUT_OF_RANGE	FL_RET_ERR_SPECIFIC_BASE-3

4.12 Updating Embedded Application

The eat1_09UpdateEmbeddedAp function initiates the embedded application updating procedure.

- **Prototype**

```
s32 eat1_09UpdateEmbeddedAp( u16 startID, u16 idCount, u32 osSize) ;
```

- **Parameters**

startID: The start ID user wants to store the firmware.
idCount: The ID count of the flash objects
osSize: The total size of the new embedded application

- **Return values**

FL_OK: System will begin to update the embedded application upon exiting the current application.
FL_RET_ERR_OVERSIZE: An error occurred during reading flash or when the object size is bigger than 8K byte.
FL_RET_ERR_UNMATCH: The size of the new application stored on the flash does not match the parameter osSize.

Note: After calling eat1_09UpdateEmbeddedAp, updating process does not start immediately; it will wait after current application to exit fl_entry().

Example:

```
void fl_entry()
{
    bool                keepGoing = TRUE;
    FlEventBuffer       flEventBuffer;

    /* Hardware initiation here*/
    while (keepGoing == TRUE)
    {
        eat1_02GetEvent (&flSignalBuffer);

        switch(flEventBuffer.eventTyp)
        {
            /*all flash operation will be started after this*/
            /*this event will come in when any interrupt occurs*/
            case EVENT_INTR:
            break;
```

```
                /*this event will come in when any key is pressed*/
                case EVENT_KEY:
                /*get embedded software from GPRS or other mode,
                * note, event flash ID's length should be less than 60000 bytes*/
                ...
                ...
                ebdat3_03FlashWriteData0,8192,writedatabuffer0);
                ebdat3_03FlashWriteData(1,8192,writedatabuffer1);
                ebdat3_03FlashWriteData(2,8192,writedatabuffer2);
                ...
                ...
                ebdat3_03FlashWriteData(19,8192,writedatabuffer19);
                /*in this case ,the osSize is 8192*20*/
                eat1_09UpdateEmbeddedAp(10000,20,osSize);
                /*When it exits the fl_entry, the SIMCom core software will begin to update
                   EmbeddedAp*/
                keepGoing = FALSE;
                break;

                /*this event will come in when ebdat9_03SetOutputdataToFL(TRUE) is called
                   *and infos come from SIMCom core software*/
                case EVENT_MODEMDATA:
                break;
                /*this event will come when ebdat9_04SetInputdataToFL(TRUE) is called and
                   * there are data from the serial port or the trace port*/
                case EVENT_UARTDATA:
                break;

                /*this event will come when some defined Timer expires*/
                case EVENT_TIMER:
                break;
                default:
                break;
            }
        }
}
```

Once fl_entry() exits, the update process will begin.

5 AT+CRWP

Due to the consideration of versatility, AT+CRWP allows developer to pass data in the form of AT commands. Disregarding ebdat9_03SetModemdataToFL setting, string after “AT” will be passed to embedded application through **EVENT_MODEMDATA**, developer can parse the string that suits their specification.

Following example represents the basic idea of how to parse attached string and apply customer rules.

```
/*at command (at+crwp) is the command string which will fill in the struct
   outputdata_evt.data */
if(flEventBuffer.event_p.outputdata_evt.type == MODEM_CRWP)
{
      Int8 para1=0;
      Int16 para3=0;
      Int8 para2=2;
     sscanf(strchr(flEventBuffer.event_p.modemdata_evt.data,'=')+1,"%d,%d,%d",&para1,
            &para2,&para3);
      switch(para1)
      {   /*get the first para, then decide which branch it will go*/
          case 0:
          ebdat9_02SendToSerialPort("play audio\x0d\x0a",12);
          ebdat10_01PlayContinousAudio (para2);
          break;
          case 1:
          ebdat9_02SendToSerialPort("stop audio\x0d\x0a",12);
          ebdat10_02StopContinousAudio();
          break;
          /*GPIO operation example */
          case 2:
          break;
          …
    }
}
```

Developer can establish their strings parsing rules freely, in this case, it takes three integers after the char “=”, and assign them to variable para1, para2, and para3 accordingly.

```
Sscanf(strchr(flEventBuffer.event_p.modemdata_evt.data,'=')+1,"%d,%d,%d",&para1,
&para2,&para3);
```

The AT command at input terminal can look like:
AT+CRWP=1,2,1, while “AT+CRWP=1,2,1” is passed to embedded application.

Appendix A: SIMCom module pins

The following table is PIN mapping of SIM900 and SIM900A.

SIM900 and SIM900A H/W SPEC.		Embedded-AT Interface		
Pin NO.	**Pin Name**	**Default Function**	**Multi Function**	**GPIO**
3	UART_DTR	UART_DTR		I
4	UART_RI	UART_RI	GPIO	I/O
5	UART_DCD	UART_DCD	GPIO	I/O
6	UART_DSR	UART_DSR	GPIO	I/O
11	SPI_CLK	GPIO	SPI_CLK	I/O
12	SPI_DATA	GPIO	SPI_DATA	I/O
13	SPI_DC	GPIO	SPI_DC	I/O
14	SPI_CS	GPIO	SPI_CS	I/O
34	SIM_PRES	SIM_PRES	GPIO	I/O
37	I2C_SDA	GPIO	/INTR	I/O
38	I2C_SCL	GPIO	/INTR	I/O
40	KBR4/GPIO1	GPIO1	KBR4	I/O
41	KBR3/GPIO2	GPIO2	KBR3	I/O
42	KBR2/GPIO3	GPIO3	KBR2	I/O
43	KBR1/GPIO4	GPIO4	KBR1	I/O
44	KBR0/GPIO5	GPIO5	KBR0	I/O
47	KBC4/GPIO6	GPIO6	KBC4	I/O
48	KBC3/GPIO7	GPIO7	KBC3	I/O
49	KBC2/GPIO8	GPIO8	KBC2	I/O
50	KBC1/GPIO9	GPIO9	KBC1	I/O
51	KBC0/GPIO10	GPIO10	KBC0	I/O
52	NETLIGHT	NETLIGHT	GPIO	I/O
66	STATUS	STATUS	GPIO	I/O
67	GPIO11	GPIO11	/INTR	I/O
68	GPIO12	RING	/INTR/ GPIO	I/O

Appendix B: Example

SIMCom provides some examples such as CSD, FCM, GPIO, HTTP, SMS, SPI, SYSTEM API and TIMER. In these examples, users can learn how to create their own project and how to write their own code.

At first user should write user's own fl_entry() function. fl_entry is the main entrance to the embedded application. Then user should call eat1_02GetEvent() to get the EVENT from the core system, as shown below:

```
/*main function */
void fl_entry()
{
    bool keepGoing = TRUE;
    while (keepGoing == TRUE)
    {
        /*get event from SIMCom Core software*/
        eat1_02GetEvent (&flEventBuffer);
        switch(flEventBuffer.eventTyp)
        {
            ……
            default:
            break;
        }
    }
}
```

User can call **ebdat9_01SendToModem ()** to send an AT command to the core system. And if user wants to receive the response of the AT command, user should call ebdat9_03SetModemdataToFL(TRUE) first. The response of the AT command will be received from eat1_02GetEvent(). The type of EVENT is EVENT_MODEMDATA and user should use union "modemdata_evt" to get the data. The type of modemdata_evt is MODEM_CMD or MODEM_CRWP which is "AT+CRWP" command from the core system as shown below:

```
/*main function */
void fl_entry()
{
    bool keepGoing = TRUE;
    FlEventBuffer    flEventBuffer;
    while (keepGoing == TRUE)
    {
        /*get event from SIMCom Core software*/
        eat1_02GetEvent (&flEventBuffer);
```

```
        switch(flEventBuffer.eventTyp)
        {
            case EVENT_MODEMDATA:
            {
                /*execute AT+CRWP    to trigger this function.*/
                if(flEventBuffer.eventData.modemdata_evt.type == MODEM_CRWP)
                {
                    /*add user's own code here*/
                }
                else if (flEventBuffer.eventData.modemdata_evt.type == MODEM_CMD)
                {
                    /*add user's own code here, to get the response of the AT command
                      from the core system.*/
                }
            }
            default:
            break;
        }
    }
}
```

If user wants to receive the data from the serial port, user should call the ebdat9_04SetUartdataToFL(TRUE) to set the UART data which is sent to the application system instead of the core system. Then if the data are received from the serial port, user calls eat1_02GetEvent() to get the data from the core system. The type of EVENT is EVENT_UARTDATA, and user should use union “uartdata_evt” to get the data. If the data are received from the UART, the type of uartdata_evt will be DATA_SERIAL. If the data are received from the debug port, the type of uartdata_evt will be DATA_DEBUG as shown below:

```
/*main function */
void fl_entry()
{
    bool keepGoing = TRUE;
    FlEventBuffer    flEventBuffer;
    while (keepGoing == TRUE)
    {
        /*get event from SIMCom Core software*/
        eat1_02GetEvent (&flEventBuffer);
        switch(flEventBuffer.eventTyp)
        {
            case EVENT_UARTDATA:
            {
```

```
                    /*execute AT+CRWP to trigger this function.*/
                    if(flEventBuffer.eventData.uartmdata_evt.type == DATA_SERIAL)
                    {
                        /*add user's own code here, these data are received from the UART*/
                    }
                    else if (flEventBuffer.eventData.uartmdata_evt.type == DATA_DEBUG)
                    {
                        /*add user's own code here. These data are received from the debug
                          port.*/
                    }
                }
                default:
                break;
            }
        }
}
```

Contact us:
Shanghai SIMCom Wireless Solutions Ltd
Add: SIM Technology Building A,
No. 633, Jinzhong Road, Shanghai, P. R. China 200335
Tel: +86 21 3252 3300
Fax: +86 21 3252 3020
URL: www.sim.com