SIM900_Embedded AT® Application Note_V1.02

Development Kit

Hardware Design

Reference Design

Software Design

Document Title:	SIM900_Embedded AT® Application Note
Version:	1.02
Date:	2013-01-09
Status:	Released
Document Control ID:	SIM900_Embedded AT® Application Note_V1.02

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

This document contains proprietary technical information which is the property of SIMCom Limited., copying of this document and giving it to others and the using or communication of the contents thereof, are forbidden without express authority. Offenders are liable to the payment of damages. All rights reserved in the event of grant of a patent or the registration of a utility model or design. All specification supplied herein are subject to change without notice at any time.

Copyright © Shanghai SIMCom Wireless Solutions Ltd. 2013

Contents

Contents 3
Version history 9
1. Introduction 10
1.1 Purpose 10
1.2 Coding style 10
1.3 References 10
1.4 Glossary 10
1.5 Abbreviations 11
2 Description 12
2.1 Software Architecture 12
2.1.1 Software Organization 12
2.1.2 Resource supplied by SIMCom 13
2.1.3 Software Supplied by SIMCom 13
2.2 Minimum Embedded Application Code 13
2.3 fl_entry() 14
2.4 fl_MultiTaskPrio1() 14
2.5 fl_MultiTaskPrio2() 14
2.6 fl_MultiTaskPrio3() 15
2.7 fl_MultiTaskPrio4() 15
2.8 fl_MultiTaskPrio5() 15
2.9 Embedded AT Memory resources 15
3 EVENT 16
3.1 EVENT Type 16
3.1.1 FlEventType 16
3.1.2 EVENT_INTR 17
3.1.3 EVENT_KEY 17
3.1.4 EVENT_UARTDATA 17
3.1.5 EVENT_MODEMDATA 17
3.1.6 EVENT_TIMER 17
3.1.7 EVENT_SERIALSTATUS 18
3.1.8 EVENT_SOCKET 18
3.1.9 EVENT_DTMF(*) 18
3.1.10 EVENT_UART_READY 18
3.1.11 EVENT_MODEM_APDU 18
3.1.12 EVENT_SIMCARD_APDU 18
3.1.13 EVENT_FLASH_READY 19
3.1.14 EVENT_MSG 19
3.1.15 EVENT_SMS_IND 19
3.1.16 EVENT_CREG_IND 19
3.1.17 EVENT_CGREG_IND 19

3.1.18 Example 19
3.2 EVENT Data 20
3.2.1 EventData 20
3.2.2 TIMER_EVT....21
3.2.3 KEY_EVT....21
3.2.4 UARTDATA_EVT 21
3.2.5 MODEMDATA_EVT 22
3.2.6 INTR_EVT....23
3.2.7 SERIALSTATUS_EVT....23
3.2.8 SOCKETEVENT_EVT 24
3.2.9 DTMF_EVENT....24
3.2.10 UARTREADYEVENT_EVT....25
3.2.11 MODEMAPDU_EVT....25
3.2.12 SIMCARDAPDU_EVT 27
3.2.13 MSG_EVT 28
3.2.14 SMSIND_EVT....28
3.2.15 CREGIND_EVT 29
3.2.16 EVENT_CGREG_IND 29
3.2.17 Examples....30
4 API 31
4.1 Data Types....31
4.2 System API....31
4.2.1 eat1_02GetEvent/eat_GetEvent 31
4.2.2 ebdat4_03Reset/eat_Reset....32
4.2.3 ebdat4_04Wdtkick/eat_Wdtkick 32
4.2.4 ebdat4_05PowerDown/eat_PowerDown....33
4.2.5 eat1_09UpdateEmbeddedAp/ eat_UpdateEmbeddedAp....33
4.2.6 ebdat6_17DisablePowerOffKey/eat_DisablePowerOffKey....33
4.2.7 ebdat6_18EnablePowerOffKey/eat_EnablePowerOffKey....33
4.3 FLASH API....34
4.3.1 ebdat3_05FlashGetLen/eat_FlashGetLen 34
4.3.2 ebdat3_06FlashDelete/eat_FlashDelete 34
4.3.3 ebdat3_07FlashGetFreeSize/eat_FlashGetFreeSize....35
4.3.4 ebdat3_03FlashWriteData/eat_FlashWriteData 35
4.3.5 ebdat3_04FlashReadData/eat_FlashReadData....36
4.3.6 ebdat3_08FlashFileRead/eat_FlashFileRead 36
4.3.7 ebdat3_09FlashFileWrite/eat_FlashFileWrite....37
4.3.8 ebdat3_10FlashFileDelete/eat_FlashFileDelete....38
4.3.9 ebdat3_11FlashFileGetLen/eat_FlashFileGetLen....38
4.4 Periphery API....38
4.4.1 Module Pins 39
4.4.1.1 FlPinName....39
4.4.1.2 FlPinMode....39
4.4.2 Periphery functions 39

4.4.2.1 ebdat6_08pinConfigureToUnused/eat_pinConfigureToUnused 40
4.4.2.2 ebdat6_06QueryPinMode/eat_QueryPinMode 40
4.4.3 Periphery-SPI 41
4.4.3.1 ebdat5_01SpiConfigure/eat_SpiConfigure 41
4.4.3.2 ebdat5_02SpiWriteByte/eat_SpiWriteByte 43
4.4.3.3 ebdat5_03SpiReadByte/eat_SpiReadByte 43
4.4.3.4 ebdat5_04SpiWriteBytes/eat_SpiWriteBytes 43
4.4.3.5 ebdat5_21EnhanceSpiConfigure/eat_EnhanceSpiConfigure 44
4.4.3.6 ebdat5_22EnhanceSpiWriteByte/eat_EnhanceSpiWriteByte..................46
4.4.3.7 ebdat5_23EnhanceSpiReadByte/eat_EnhanceSpiReadByte 46
4.4.3.8 ebdat5_24EnhanceSpiWriteBytes/eat_EnhanceSpiWriteBytes 47
4.4.3.9 ebdat5_25EnhanceSpiReadBytes/eat_EnhanceSpiReadBytes 47
4.4.4 Periphery-Display 48
4.4.4.1 ebdat05_11DispConfig/eat_DispConfig 48
4.4.4.2 ebdat05_12DispWriteCommand/eat_DispWriteCommand 49
4.4.4.3 ebdat05_13DispWriteData/eat_DispWriteData......................................50
4.4.5 Periphery interrupt 50
4.4.5.1 ebdat6_13IntSubscribe/eat_IntSubscribe 50
4.4.6 Periphery square wave 51
4.4.6.1 ebdat6_19SqWaveSubscribe/eat_SqWaveSubscribe...............................51
4.4.6.2 ebdat6_20SqWaveUnsubscribe/eat_SqWaveUnsubscribe 52
4.4.7 Periphery-GPIO...53
4.4.7.1 ebdat6_02GpioSubscribe/eat_GpioSubscribe 53
4.4.7.2 ebdat6_05ReadGpio/eat_ReadGpio 54
4.4.7.3 ebdat6_04WriteGpio/eat_WriteGpio...54
4.4.7.4 ebdat6_27SetWatchDogGpio/eat_SetWatchDogGpio 54
4.4.8 Periphery-Keypad 55
4.4.8.1 ebdat6_15KeySubscribe/eat_KeySubscribe 55
4.4.9 Periphery-I2C 55
4.4.9.1 ebdat15_01I2C_SpeedConfig/eat_I2C_SpeedConfig 55
4.4.9.2 ebdat15_02I2C_ReadWriteDone/eat_I2C_ReadWriteDone 56
4.4.9.3 ebdat15_03I2C_GetStatus/eat_I2C_GetStatus.......................................56
4.4.9.4 ebdat15_04I2C_SetStatus/eat_I2C_SetStatus 57
4.4.9.5 ebdat15_05I2C_INITIALIZE_TRANSFER/eat_I2C_INITIALIZE_TRANSFER 57
4.4.9.6 ebdat15_06I2C_PUT_DATA/eat_I2C_PUT_DATA 58
4.4.9.7 ebdat15_07I2C_GET_DATA/eat_I2C_GET_DATA 58
4.5 Audio API 59
4.5.1 ebdat10_01PlayContinousAudio/eat_PlayContinousAudio...........................59
4.5.2 ebdat10_02StopContinousAudio/eat_StopContinousAudio 59
4.5.3 ebdat10_03PlaySingleAudio/eat_PlaySingleAudio 59
4.5.4 ebdat10_04PlaySingleAudioFromFile/eat_PlaySingleAudioFromFile 60
4.5.5 ebdat10_07PlayRemoteAmrFromFile/eat_PlayRemoteAmrFromFile 60

4.5.6 AUDIO TRACKS 61
4.6 TIMER API 63
4.6.1 Timer structure 63
4.6.2 ebdat8_01StartTimer/eat_StartTimer 63
4.6.3 ebdat8_02StopTimer/eat_StopTimer 64
4.6.4 ebdat8_04SecondToTicks/eat_SecondToTicks 64
4.6.5 ebdat8_05MillisecondToTicks/eat_MillisecondToTicks 65
4.6.6 ebdat8_03GetRelativeTime/eat_GetRelativeTime 65
4.6.7 ebdat8_06GetSystemTime/eat_GetSystemTime 65
4.6.8 ebdat8_08GetSystemTickCounter/eat_GetSystemTickCounter 66
4.6.9 ebdat8_10CurrentTaskSleep/eat_CurrentTaskSleep 66
4.7 FCM API 68
4.7.1 ebdat9_01SendToModem/eat_SendToModem 69
4.7.2 ebdat9_02SendToSerialPort/eat_SendToSerialPort 69
4.7.3 ebdat9_03SetModemdataToFL/eat_SetModemdataToFL 70
4.7.4 ebdat9_04SetUartdataToFL/eat_SetUartdataToFL 70
4.7.5 ebdat9_05GetSerialPortTxStatus/eat_GetSerialPortTxStatus 70
4.7.6 ebdat6_23GetRTSPinLevel/eat_GetRTSPinLevel 71
4.7.7 ebdat9_09ChangeMainUartBaudRate/eat_ChangeMainUartBaudRate 71
4.7.8 ebdat9_10GetMainUartBaudRate/eat_GetMainUartBaudRate 72
4.7.9 ebdat9_11ChangeMainUartDataFormat/eat_ChangeMainUartDataFormat ... 72
4.7.10 ebdat9_12GetMainUartDataFormat/eat_GetMainUartDataFormat 73
4.7.11 ebdat9_13ChangeMainUartFlowControl/eat_ChangeMainUartFlowControl 73
4.7.12 ebdat9_14GetMainUartFlowControl/eat_GetMainUartFlowControl 74
4.7.13 ebdat9_15SubscribeURC/eat_SubscribeURC 74
4.7.14 ebdat9_16UnSubscribeURC/eat_UnSubscribeURC 75
4.7.15 ebdat9_17GetURCNum/eat_GetURCNum 75
4.7.16 ebdat9_19SubscribeATCommand/eat_SubscribeATCommand 76
4.7.17 ebdat9_20UnsubscribeATCommand/eat_UnsubscribeATCommand 76
4.7.18
ebdat9_24MainUartPortIsTransmitterEmpty/eat_MainUartPortIsTransmitterE
mpty 77
4.8 Debug API 77
4.8.1 ebdat7_00EnterDebugMode/eat_EnterDebugMode 77
4.8.2 ebdat7_01DebugTrace/eat_DebugTrace 78
4.8.3 ebdat7_02DebugUartSend/eat_DebugUartSend 78
4.9 Other API 79
4.9.1 ebdat4_22GetADCValue/eat_GetADCValue 79
4.9.2 ebdat4_23GetBatteryVoltage/eat_GetBatteryVoltage 79
4.9.3 ebdat4_24GetModuleTemperature/eat_GetModuleTemperature 80
4.9.4 ebdat4_25GetRegistrationStatus/eat_GetRegistrationStatus 80
4.9.5 ebdat4_26GetGPRSRegistrationStatus/eat_GetGPRSRegistrationStatus 81
4.9.6 ebdat4_27GetGPRSAttachStatus/eat_GetGPRSAttachStatus 81
4.9.7 ebdat4_28GetCSQValue/eat_GetCSQValue 82

4.9.8 ebdat4_29GetServiceCellInformation/eat_GetServiceCellInformation..........82
4.9.9 ebdat4_30GetNeighborCellInformation/eat_GetNeighborCellInformation....83
4.9.10 ebdat4_31GetIMEI/eat_GetIMEI..........83
4.9.11 ebdat4_32GetCfunValue/eat_GetCfunValue..........84
4.9.12 ebdat4_33GetModuleCpinStatus/eat_GetModuleCpinStatus..........84
4.9.13 ebdat4_35GetSIMCardIMSI/eat_GetSIMCardIMSI..........85
4.9.14 ebdat4_36GetSIMCardICCID/eat_GetSIMCardICCID..........86
4.9.15 ebdat4_37GetSIMCardSPN/eat_GetSIMCardSPN..........86
4.9.16 ebdat4_38SetSMSIndEvent/eat_SetSMSIndEvent..........86
4.9.17 ebdat4_39SetCregIndEvent/eat_SetCregIndEvent..........87
4.9.18 ebdat4_40SetCgregIndEvent/eat_SetCgregIndEvent..........87
4.9.19 ebdat4_34GetCurrentTaskID/eat_GetCurrentTaskID..........87
4.10 Standard library API..........88
4.10.1 Standard input/output functions..........88
4.10.2 ebdat4_10strRemoveCRLF/eat_strRemoveCRLF..........88
4.10.3 ebdat4_11strGetParameterString/eat_strGetParameterString..........89
4.10.4 ebdat6_17DisablePowerOffKey/eat_DisablePowerOffKey..........89
4.10.5 ebdat6_18EnablePowerOffKey/eat_EnablePowerOffKey..........90
4.10.6 ebdat4_15ExitOutOfSleepMode/eat_ExitOutOfSleepMode..........90
4.10.7 ebdat4_17EnterSleepMode/eat_EnterSleepMode..........90
4.11 SOCKET API..........91
4.11.1 ebdat11_10GprsActive/eat_GprsActive..........91
4.11.2 ebdat11_15GprsDeactive/eat_GprsDeactive..........91
4.11.3 ebdat11_20SocketConnect/eat_SocketConnect..........92
4.11.4 ebdat11_25SocketClose/eat_SocketClose..........92
4.11.5 ebdat11_30SocketSend/eat_SocketSend..........93
4.11.6 ebdat11_35SocketRecv/eat_SocketRecv..........94
4.11.7 ebdat11_45SocketTcpServerSet/eat_SocketTcpServerSet..........94
4.11.8 ebdat11_50GetLocalIpAddr/eat_GetLocalIpAddr..........95
4.12 Error Codes..........95
4.13 Updating Embedded Application/eat_UpdateEmbeddedAp..........96
4.14 DTMF API..........98
4.14.1 ebdat10_06DTMFDetectEnable/eat_DTMFDetectEnable..........98
4.15 SIM card API..........99
4.15.1 ebdat13_00SetModemAPDUToFL/eat_SetModemAPDUToFL..........99
4.15.2 ebdat13_01SetSIMCardAPDUToFL/eat_SetSIMCardAPDUToFL..........99
4.15.3 ebdat13_03SendResetReqToSIMCard/eat_SendResetReqToSIMCard..........100
4.15.4 ebdat13_05SendSIMCardResetCnfToModem/eat_SendSIMCardResetCnfToModem 100
4.15.5 ebdat13_08SendAPDUReqToSIMCard/eat_SendAPDUReqToSIMCard..........100
4.15.6 ebdat13_10SendAPDUCnfToModem/eat_SendAPDUCnfToModem..........101
4.15.7 ebdat13_11SoftSendAPDUCnfToModem/eat_SoftSendAPDUCnfToModem 101

4.16 Multi task API 102
4.16.1 ebdat4_21SendEventMsg/eat_SendEventMsg 102
4.16.2 ebdat14_00CreateSem/eat_CreateSem 102
4.16.3 ebdat14_01semPend/eat_semPend 103
4.16.4 ebdat14_02semPost/eat_semPost 103
5 AT+CRWP 104
Appendix A: SIMCom module pins 105
Appendix B: Example 106

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
2013-1-8	V1.02	1. Add new API function 2. Modify ebdat05_11DispConfig function 3. Add I2C function 4. Add function point	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.
It also supplies 6 tasks and 6 semaphores. These 6 tasks are EAT_TASK, MULTI_TASK_1, MULTI_TASK_2, MULTI_TASK_3, MULTI_TASK_4, MULTI_TASK_5. In all of them, the EAT_TASK is the main task. All of the events will be only sent to the EAT_TASK.
In order to make the communication between tasks, Embedded AT also supplies the functions which send message to other tasks and some semaphore function.

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_03Reset", 4 is the file name index part, and 03 is the function number of the file. It is very easy for the user or the SIMCom developers to trace problems this way.
And SIMCom supplies the function point API which is used when customers do not want to compile their app again after the core is changed.
Note: If customers do not want to compile their app again when the core is changed., only function point APIs can be called otherwise the module will reset when the core is changed. Please refer to <u>Char. 4</u>.

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:

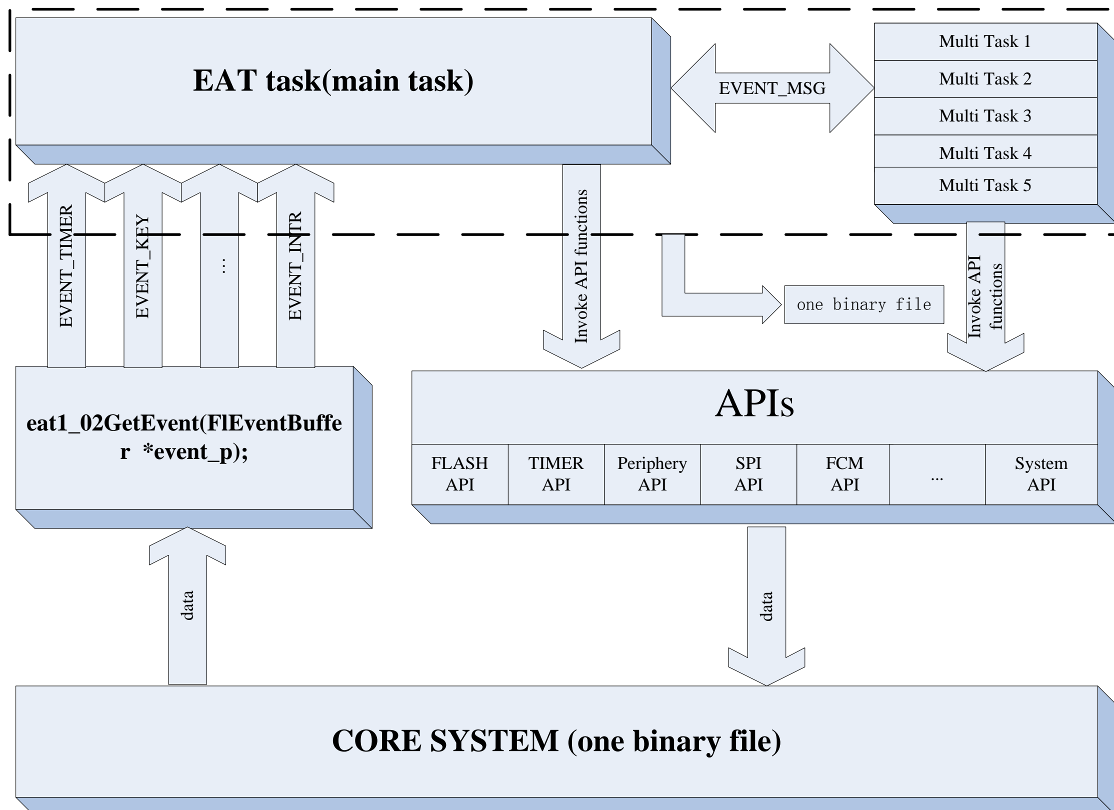


Figure 1: General software architecture

Information flow

In figure 1, the app code has 6 tasks: Eat task, multi task1, multi task 2, multi task 3, multi task 4 and multi task 5.

➢ Communication between core system and tasks:

Only Eat task (main task) can receive the EVENTs from the core system. When the core system sends a message with categorized EVENT type to the EAT task, the eat1_02GetEvent function will catch it.

When the tasks send 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.

- Communications among tasks:

  To send a message from a source task to a destination task, developer can simply invokes API function ebdat4_21SendEventMsg in the source task, and then the eat1_02GetEvent function will catch this EVENT_MSG event and forward it to the destination task.

  Any task can receive EVENT_MSG event from other tasks or itself.

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(). The following 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 fl_MultiTaskPrio1()

fl_MultiTaskPrio1() is the entrance function of Multi task 1, it works almost as main() in standard C application. Embedded application quits upon exiting fl_MultiTaskPrio1(). The following example uses a while statement to keep the application alive until the application developer ends it by setting keepGoing = FALSE; If this task is not used, this function must be reserved and keep it going.

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

2.5 fl_MultiTaskPrio2()

fl_MultiTaskPrio2() is the entrance function of Multi task 2, it works almost as main() in standard C application. Embedded application quits upon exiting fl_MultiTaskPrio2().The following example uses a while statement to keep the application alive until the application developer ends it

by setting keepGoing = FALSE; If this task is not used, this function must be reserved and keep it going.

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

2.6 fl_MultiTaskPrio3()

fl_MultiTaskPrio3() is the entrance function of Multi task 3, it works almost as main() in standard C application. Embedded application quits upon exiting fl_MultiTaskPrio3().The following example uses a while statement to keep the application alive until the application developer ends it by setting keepGoing = FALSE; If this task is not used, this function must be reserved and keep it going.

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

2.7 fl_MultiTaskPrio4()

fl_MultiTaskPrio4() is the entrance function of Multi task 4, it works almost as main() in standard C application. Embedded application quits upon exiting fl_MultiTaskPrio4().The following example uses a while statement to keep the application alive until the application developer ends it by setting keepGoing = FALSE; If this task is not used, this function must be reserved and keep it going.

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

2.8 fl_MultiTaskPrio5()

fl_MultiTaskPrio5() is the entrance function of Multi task 5, it works almost as main() in standard C application. Embedded application quits upon exiting fl_MultiTaskPrio5().The following example uses a while statement to keep the application alive until the application developer ends it by setting keepGoing = FALSE; If this task is not used, this function must be reserved and keep it going.

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

2.9 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_UART_READY,
	EVENT_MODEM_APDU,
	EVENT_SIMCARD_APDU,
	EVENT_FLASH_READY,
	EVENT_MSG,
	EVENT_SMS_IND,
	EVENT_CREG_IND,
	EVENT_CGREG_IND,
	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 EVENT_DTMF(*)

The event can be triggered when DTMF function is enabled (see chapter 4.1 for details).
Note: This event is only existed in DTMF firmware. It is not supported in normal version.

3.1.10 EVENT_UART_READY

The event is triggered when the serial port is ready and AT command is ready. When the module is powered on, the application code should wait for this event and then data can be sent to the core or the serial port.

3.1.11 EVENT_MODEM_APDU

The event is triggered when the core requests data from SIM card. This event is from the core.
Important Remarks:
In order to get the data which is sent from the module to the SIM card, the function ebdat13_00SetModemAPDUToFL (TRUE) has to be set (default is FALSE) before embedded application can capture the data from the core.

3.1.12 EVENT_SIMCARD_APDU

The event is triggered when the SIM card responds to the module. This event is from the SIM card.
Important Remarks:
In order to get data which the SIM card responds to the module, the function ebdat13_01SetSIMCardAPDUToFL (TRUE) has to be set (default is FALSE) before embedded application can capture data from the SIM card.

3.1.13 EVENT_FLASH_READY

The event is triggered when the file system is ready. When the module is powered on, the application code should wait for this event and then the data can be written to the flash.

3.1.14 EVENT_MSG

The event is received when the current task receives a message from other tasks or itself.

3.1.15 EVENT_SMS_IND

This event will be triggered when a short message is received.

Important Remarks:

- The same situation in EVENT_UARTDATA applies here too, the function ebdat4_38SetSMSIndEvent (TRUE) has to be set (default is FALSE) if the EVENT_SMS_IND is wanted to receive.

3.1.16 EVENT_CREG_IND

This event will be triggered when CREG value is changed. It is the same as the AT command "AT+CREG=1".

Important Remarks:

- The same situation in EVENT_UARTDATA applies here too, the function ebdat4_39SetCregIndEvent (TRUE) has to be set (default is FALSE) if the EVENT_CREG_IND is wanted to receive.

3.1.17 EVENT_CGREG_IND

This event will be triggered when CGREG value is changed. It is the same as the AT command "AT+CGREG=1".

Important Remarks:

- The same situation in EVENT_UARTDATA applies here too, the function ebdat4_40SetCgregIndEvent (TRUE) has to be set (default is FALSE) if the EVENT_CGREG_IND is wanted to receive.

3.1.18 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 the event associated to its 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;
      DTMF_EVENT     dtmf_evt;
      UARTREADYEVENT_EVT uartReady_evt;
      MODEMAPDU_EVT modemAPDU_evt;
      SIMCARDAPDU_EVT simcardAPDU_evt;
      MSG_EVT     msg_evt;
      SMSIND_EVT         smsInd_evt;
      CREGIND_EVT       cregInd_evt;
      CGREGIND_EVT cgregInd_evt;
}EventData;}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 DTMF_EVENT

```
typedef struct DTMF_EVENTTag
{
      ascii demfChar;
      u8 reserve[3];
}DTMF_EVENT;
```

demfChar: The character of DTMF.

Note: This event is only existed in DTMF firmware. It is not supported in normal version.

3.2.10 UARTREADYEVENT_EVT

```
typedef struct UARTREADYEVENT_EVTTag
{
    u32 active;
}UARTREADYEVENT_EVT;
```

active: 0: serial port is ready.
1: serial port is not ready.

3.2.11 MODEMAPDU_EVT

```
typedef struct MODEMAPDU_DATATag
{
    u8  v_Class;
    u8  v_Instruction;
    u8  v_P1;
    u8  v_P2;
    u8  v_P3;
    u8  v_unused1;
    u8  v_unused2;
    u8  v_unused3;
    u8  a_CData[256];
}MODEMAPDU_DATA;
typedef enum FlModemAPDUTypeTag
{
    FL_MOD_APDU_RESET,
    FL_MOD_APDU_DISCONNECT,
    FL_MOD_APDU_REQ_DATA,
    FL_MOD_APDU_SEND_DATA,
    FL_MOD_APDU_MAX
}FlModemAPDUType;
typedef struct MODEMAPDU_EVTTag
{
    MODEMAPDU_DATA apduData;
    u8    v_errorValue;
    u8    v_resetType;
    FlModemAPDUType apduType;
}MODEMAPDU_EVT;
```

apduData: The APDU data which is requested by the core.

MODEMAPDU_DATA

v_Class : APDU Command Class: Should always be 0xA0

v_Instruction: APDU Command Type: As defined in GSM 11.11, chapter 9.2

v_P1: APDU P1 byte coding: As defined in GSM 11.11, chapter 9.2

v_P2: APDU P2 byte coding: As defined in GSM 11.11, chapter 9.2

v_P3: APDU P3 byte coding: As defined in GSM 11.11, chapter 9.2

a_CData: APDU command data: needs only to be provided in case of commands which imply data sending to the card (Selection, file update, GSM algo,…). For such commands, nb(? number) of bytes to be sent is coded in P3 byte.

v_errorValue: Error type

v_resetType: Reset type, it is available when **apduType** is **FL_MOD_APDU_RESET**.

apduType: The type of the APDU data

FlModemAPDUType

FL_MOD_APDU_RESET: Reset the SIM card.

FL_MOD_APDU_DISCONNECT: Disconnect the SIM card.

FL_MOD_APDU_REQ_DATA: Request data from the SIM card.

FL_MOD_APDU_SEND_DATA: Only send data to the SIM card.

3.2.12 SIMCARDAPDU_EVT

```
typedef struct SIMCARDAPDU_DATATag
{
    u16   v_len;
    u8    a_RData[258];
    u8    v_unused1;
    u8    v_unused2;
    u8    v_unused3;
    u8    v_unused4;
}SIMCARDAPDU_DATA;

typedef struct SIMCARDRESET_CNFTag
{
    u8  v_VoltageValue;
    u8  a_AnswerToReset[33];
    u8  unused1;
    u8  unused2;
}SIMCARDRESET_CNF;

typedef enum FlSIMCardAPDUTypeTag
{
    FL_SIM_APDU_INTERFACE_ERROR,
    FL_SIM_APDU_CMD_ERROR,
    FL_SIM_APDU_DATA,
    FL_SIM_ADPU_RESET_CNF,
    FL_SIM_APDU_MAX
}FlSIMCardAPDUType;

typedef struct SIMCARDAPDU_EVTTag
{
    SIMCARDAPDU_DATA apduData;
    u8    v_errorValue;
    SIMCARDRESET_CNF    resetCnf;
    FlSIMCardAPDUType apduType;
}SIMCARDAPDU_EVT;
```

apduData: The APDU data from the SIM card.

SIMCARDAPDU_DATA

v_len: APDU response length: Length of data provided in SIM APDU response (if any) (Expressed in bytes)

a_RData: APDU response data: needs only to be provided in case of commands which imply data receiving from the SIM card (Get response, file reading, Status, …).

v_errorValue: The Error type

resetCnf: Reset SIM card confirmation

SIMCARDRESET_CNF

v_VoltageValue： The voltage of the SIM card.

0: the voltage is 5V

1: the voltage is 3V

2: the voltage is 1.8V

a_AnswerToReset: ATR data bytes

apduType： The type of the APDU data.

FlSIMCardAPDUType

FL_SIM_APDU_INTERFACE_ERROR: The length of the response is incorrect.

FL_SIM_APDU_CMD_ERROR: The command is incorrect.

FL_SIM_APDU_DATA: The APDU data from the SIM card.

FL_SIM_ADPU_RESET_CNF: The ATR data from the SIM card.

3.2.13 MSG_EVT

```
typedef enum FLMsgTaskIDTag
{
    FL_EAT_TASK,
    FL_MULTI_TASK_1,
    FL_MULTI_TASK_2,
    FL_MULTI_TASK_3,
    FL_MULTI_TASK_4,
    FL_MULTI_TASK_5,
    NUM_OF_FL_MULTI_TASK
}FLMsgTaskID;
typedef struct MSG_EVTTag
{
    FLMsgTaskID source;
    FLMsgTaskID destination;
    u8 dataBuffer[2048];
}MSG_EVT;
```

source: the source of the message which it is sent from.

destination: the destination of the message which it is sent to.

dataBuffer: the content of the message.

3.2.14 SMSIND_EVT

```
typedef struct SMSIND_EVTTag
{
    u16 index;
}SMSIND_EVT;
```

index: the index of the short message. It is the same as the index of “+CMTI: <mem>,<index>”.

3.2.15 CREGIND_EVT

```
typedef struct CREGIND_EVTTag
{
    u8 status;
}CREGIND_EVT;
```

status: the status of ME network registration.

0: Not registered, MT is not currently searching a new operator to register to
1: Registered, home network
2: Not registered, but MT is currently searching a new operator to register to
3: Registration denied
4: Unknown
5: Registered, roaming

3.2.16 EVENT_CGREG_IND

```
typedef struct CGREGIND_EVTTag
{
    u8 status;
}CGREGIND_EVT;
```

status: the status of ME GPRS network registration.

0: not registered, mt is not currently searching an operator to register to.the gprs service is disabled, the ue is allowed to attach for gprs if requested by the user.
1: registered, home network.
2: not registered, but mt is currently trying to attach or searching an operator to register to. the gprs service is enabled, but an allowable plmn is currently not available. the ue will start a gprs attach as soon as an allowable plmn is available.
3: registration denied. The gprs service is disabled, the ue is not allowed to attach for gprs if it is requested by the user.
4: unknown
5: registered, roaming

3.2.17 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.
There are two kinds of functions. One is function, and the other is function pointer which is pointed to the function. When a new core file is released by SIMCOM, if the user does not want to compile user's app file again, the user should use function pointer.
Note: If user does not want to compile his app file again when a new core is released, in user's code, a global macro "USE_C_STANDARD_LIBS" should be defined in user's project.

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, the head file needs to be included.

4.2.1 eat1_02GetEvent/eat_GetEvent

The eat1_02GetEvent/eat_GetEvent 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);
void (*const eat_GetEvent)(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_03Reset/eat_Reset

The ebdat4_03Reset/eat_Reset function resets the system. Use this function cautiously. It is not recommended to use this function generally.

- **Prototype**

```
void ebdat4_03Reset(void);
void (*const eat_Reset)(void);
```

4.2.3 ebdat4_04Wdtkick/eat_Wdtkick

The ebdat4_04Wdtkick/eat_Wdtkick 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);
void (*const eat_Wdtkick)(void);
```

4.2.4 ebdat4_05PowerDown/eat_PowerDown

The ebdat4_05PowerDown/eat_PowerDown 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);
void (*const eat_PowerDown)(void);
```

4.2.5 eat1_09UpdateEmbeddedAp/ eat_UpdateEmbeddedAp

See 4.11 Updating Embedded Application.

4.2.6 ebdat6_17DisablePowerOffKey/eat_DisablePowerOffKey

The ebdat6_17DisablePowerOffKey/eat_DisablePowerOffKey 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);
void (*const eat_DisablePowerOffKey)(void);
```

4.2.7 ebdat6_18EnablePowerOffKey/eat_EnablePowerOffKey

The ebdat6_18EnablePowerOffKey/eat_EnablePowerOffKey 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);
void (*const eat_EnablePowerOffKey)(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/eat_FlashGetLen

The ebdat3_05FlashGetLen/eat_FlashGetLen function gets the length of a specific flash.

- **Prototype**

```
s32 ebdat3_05FlashGetLen(u16 ID,u16* len);
s32 (*const eat_FlashGetLen)(u16 ID,u16*    length);
```

- **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/eat_FlashDelete

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

- **Prototype**

```
s32 ebdat3_06FlashDelete(u16 ID);
s32 (*const eat_FlashDelete)(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/eat_FlashGetFreeSize

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

- **Prototype**

```
s32 ebdat3_07FlashGetFreeSize( u32 *freeSize);
s32 (*const eat_FlashGetFreeSize)(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/eat_FlashWriteData

The ebdat3_03FlashWriteData/eat_FlashWriteData 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 );
s32 (*const eat_FlashWriteData)(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/eat_FlashReadData

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

- **Prototype**

```
s32 ebdat3_04FlashReadData(u16 ID, u16 len, u8 * data );
s32 (*const eat_FlashReadData)(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/eat_FlashFileRead

The ebdat3_08FlashFileRead/eat_FlashFileRead 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);
s32 (*const eat_FlashFileRead) (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/eat_FlashFileWrite

The ebdat3_09FlashFileWrite/eat_FlashFileWrite 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);
s32 (*const eat_FlashFileWrite)(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/eat_FlashFileDelete

The ebdat3_10FlashFileDelete/eat_FlashFileDelete 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);
s32 (*const eat_FlashFileDelete)(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/eat_FlashFileGetLen

The ebdat3_11FlashFileGetLen/eat_FlashFileGetLen function gets the length of a file.

- **Prototype**

```
s32 ebdat3_11FlashFileGetLen(u8* fileName,u16*  length);
s32 (*const eat_FlashFileGetLen)(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/eat_pinConfigureToUnused

The ebdat6_08pinConfigureToUnused/eat_pinConfigureToUnused 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);
s32 (*const eat_pinConfigureToUnused)(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/eat_QueryPinMode

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

- **Prototype**

```
s32 ebdat6_06QueryPinMode(FlPinName pinName,
                          FlPinMode *pinMode_p,
                          FlGpioDirection *isOutputDir_p);
s32 (*const eat_QueryPinMode)(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/eat_SpiConfigure

The ebdat5_01SpiConfigure/eat_SpiConfigure 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 ) ;
s32 (*const eat_SpiConfigure)(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:

3 Wire Mode				
SPI Name	**Platform Pin Name**	**Platform GPIOs**	**Direction**	**SIM900**
MOSI	SSI_DATA	GPIO50	output	DISP_DATA
SCLK	SSI_CLK	GPIO48	output	DISP_CLK
SS	SSI_SEL0	GPIO51	output	DCD
	SSI_SEL1	GPIO52	output	DISP_CS
	SSI_SEL2	GPIO53	output	DSR

4 Wire Mode				
SPI Name	**Platform Pin Name**	**Platform GPIOs**	**Direction**	**SIM900**
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	SSI_SEL0	GPIO51	output	DCD
	SSI_SEL1	GPIO52	output	DISP_CS
	SSI_SEL2	GPIO53	output	DSR

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/eat_SpiWriteByte

The ebdat5_02SpiWriteByte/eat_SpiWriteByte function writes one byte to the SPI interface.

- **Prototype**

```
s32   ebdat5_02SpiWriteByte(u8 data);
s32 (*const eat_SpiWriteByte)(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/eat_SpiReadByte

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

- **Prototype**

```
u8 ebdat5_03SpiReadByte (void);
u8 (*const eat_SpiReadByte) (void);
```

- **Parameters**

NONE

- **Return values**

One byte read from spi

4.4.3.4 ebdat5_04SpiWriteBytes/eat_SpiWriteBytes

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

- **Prototype**

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

4.4.3.5 ebdat5_21EnhanceSpiConfigure/eat_EnhanceSpiConfigure

The ebdat5_21EnhanceSpiConfigure/eat_EnhanceSpiConfigure 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_21EnhanceSpiConfigure(SsiDevNbType                ssiDevNb,
                                 SsiModeType wireMode ,
                                 SsiClockType clkSpeed ,
                                 SsiDataPolarityType clkMode ,
                                 SsiTrfFormatType msbFirst ) ;
s32 (*const eat_EnhanceSpiConfigure)(SsiDevNbType            ssiDevNb,
                                     SsiModeType             wireMode,
                                     SsiClockType            clkSpeed,
                                     SsiDataPolarityType     clkMode ,
                                     SsiTrfFormatType        msbFirst);
```

- **Parameters**

SPI parameter is made up of following parameters.
ssiDevNb:

SSI_SLAVE0 = 0 /*SIM900 UART1 DCD PIN*/
SSI_SLAVE1 = 1 /*SIM900 UART1 DSR PIN*/
SSI_SLAVE2 = 2 /*SIM900 UART1 DISP_CS PIN*/

wireMode:

SSI_3WIRE, for 3-wire mode SPI.

SSI_4WIRE, for 4-wire mode SPI.

For SIM900:

3 Wire Mode				
SPI Name	**Platform Pin Name**	**Platform GPIOs**	**Direction**	**SIM900**
MOSI	SSI_DATA	GPIO50	output	DISP_DATA
SCLK	SSI_CLK	GPIO48	output	DISP_CLK
SS	SSI_SEL0	GPIO51	output	DCD
	SSI_SEL1	GPIO52	output	DISP_CS
	SSI_SEL2	GPIO53	output	DSR

4 Wire Mode				
SPI Name	**Platform Pin Name**	**Platform GPIOs**	**Direction**	**SIM900**
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	SSI_SEL0	GPIO51	output	DCD
	SSI_SEL1	GPIO52	output	DISP_CS
	SSI_SEL2	GPIO53	output	DSR

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.

4.4.3.6 ebdat5_22EnhanceSpiWriteByte/eat_EnhanceSpiWriteByte

The ebdat5_22EnhanceSpiWriteByte/eat_EnhanceSpiWriteByte function writes one byte to the SPI interface.

- **Prototype**

```
s32 ebdat5_22EnhanceSpiWriteByte( SsiDevNbType          ssiDevNb,u8 data);
s32   (*const eat_EnhanceSpiWriteByte) (SsiDevNbType   ssiDevNb,u8 data);
```

- **Parameters**

ssiDevNb:

SSI_SLAVE0 = 0   /*SIM900 PIN:UART1 DCD */
SSI_SLAVE1 = 1   /*SIM900 PIN:UART1 DSR */
SSI_SLAVE2 = 2   /*SIM900 PIN:UART1 DISP_CS */

data: Byte to transfer

- **Return values**

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

Note:

1. You need set to ssiDevNb to the same value as the one which ebdat5_21EnhanceSpiConfigure() has configured.

4.4.3.7 ebdat5_23EnhanceSpiReadByte/eat_EnhanceSpiReadByte

The ebdat5_23EnhanceSpiReadByte/eat_EnhanceSpiReadByte function will read one byte from the SPI interface.

- **Prototype**

```
u8 ebdat5_23EnhanceSpiReadByte ( SsiDevNbType          ssiDevNb );
u8 (*const eat_EnhanceSpiReadByte) (SsiDevNbType   ssiDevNb);
```

- **Parameters**

ssiDevNb:

SSI_SLAVE0 = 0   /*SIM900 PIN:UART1 DCD */
SSI_SLAVE1 = 1   /*SIM900 PIN:UART1 DSR */
SSI_SLAVE2 = 2   /*SIM900 PIN:UART1 DISP_CS */

Note:

1. You need set ssiDevNb to the same value as the one which ebdat5_21EnhanceSpiConfigure() has configured.

- **Return values**

One byte read from spi

4.4.3.8 ebdat5_24EnhanceSpiWriteBytes/eat_EnhanceSpiWriteBytes

The ebdat5_24EnhanceSpiWriteBytes/eat_EnhanceSpiWriteBytes function will write bytes to the SPI interface. This is a block function.

- **Prototype**

```
s32 ebdat5_24EnhanceSpiWriteBytes( SsiDevNbType ssiDevNb,u8 *p_data,u32 dataSize);
s32 (*const eat_EnhanceSpiWriteBytes)(SsiDevNbType ssiDevNb, u8 *p_data, u32 dataSize);
```

- **Parameters**

ssiDevNb:

SSI_SLAVE0 = 0 /*SIM900 PIN:UART1 DCD */
SSI_SLAVE1 = 1 /*SIM900 PIN:UART1 DSR */
SSI_SLAVE2 = 2 /*SIM900 PIN:UART1 DISP_CS */

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.

Note:
1. You need set ssiDevNb to the same value as the one which ebdat5_21EnhanceSpiConfigure() has configured.

4.4.3.9 ebdat5_25EnhanceSpiReadBytes/eat_EnhanceSpiReadBytes

The ebdat5_25EnhanceSpiReadBytes/eat_EnhanceSpiReadBytes function will read bytesfrom SPI interface. This is a block function.

- **Prototype**

```
void ebdat5_25EnhanceSpiReadBytes( SsiDevNbType ssiDevNb,u8 *p_data, int nums);
void (*const eat_EnhanceSpiReadBytes) (SsiDevNbType ssiDevNb, u8 *p_data, int nums);
```

- **Parameters**

ssiDevNb:

SSI_SLAVE0 = 0 /*SIM900 PIN:UART1 DCD */
SSI_SLAVE1 = 1 /*SIM900 PIN:UART1 DSR */
SSI_SLAVE2 = 2 /*SIM900 PIN:UART1 DISP_CS */

p_data: Pointer of data to be sent.

nums: Size of data to read. It cannot exceed 4K bytes.

- **Return values**

NONE.

Note:

1. ***You need set to ssiDevNb to the same value as the one which ebdat5_21EnhanceSpiConfigure() has configured.***

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 functions. 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_67** (used as **DISP_RST), DISP_D/C, DISP _DATA, DISP_CLK** and **DISP_CS**.

4.4.4.1 ebdat05_11DispConfig/eat_DispConfig

The ebdat05_11DispConfig/eat_DispConfig function configures display interface using SIM900 and SIM900A PINs: **FL_PIN_67** (used as **DISP_RST), DISP_D/C, DISP _DATA, DISP_CLK** and FL_PIN_14 (used as **DISP_CS)**.

- **Prototype**

```
s32 ebdat05_11DispConfig ( FlPinName    rst_gpio_num,    SsiClockType clk);
s32 (*const eat_DispConfig)(FlPinName rst_gpio_num, SsiClockType clk);
```

- **Parameters**

rst_gpio_num: The GPIO used as Reset 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/eat_DispWriteCommand

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

- **Prototype**

```
s32 ebdat05_12DispWriteCommand (u8 command);
s32 (*const eat_DispWriteCommand)(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/eat_DispWriteData

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

- **Prototype**

```
s32 ebdat05_13DispWriteData (u8 data);
s32 (*const eat_DispWriteData)(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/eat_IntSubscribe

The ebdat6_13IntSubscribe/eat_IntSubscribe 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);
s32 (*const eat_IntSubscribe)(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/eat_SqWaveSubscribe

The ebdat6_19SqWaveSubscribe/eat_SqWaveSubscribe 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);
s32 (*const eat_SqWaveSubscribe)(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/eat_SqWaveUnsubscribe

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

- **Prototype**

s32 ebdat6_20SqWaveUnsubscribe(FlPWM pwm) ;

```
s32 (*const eat_SqWaveUnsubscribe)(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/eat_GpioSubscribe

The **ebdat6_02GpioSubscribe/eat_GpioSubscribe** function subscribes pins to GPIO pins and changes pin mode to **FL_PIN_MODE_GPIO**.

- **Prototype**

```
s32 ebdat6_02GpioSubscribe(FlPinName pinName,
                           FlGpioDirection gpioDir,
                           bool defValue);
s32 (*const eat_GpioSubscribe)(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/eat_ReadGpio

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

- **Prototype**

```
s32 ebdat6_05ReadGpio(FlPinName pinName, bool *inputValue_p);
s32 (*const eat_ReadGpio)(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/eat_WriteGpio

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

- **Prototype**

```
s32 ebdat6_04WriteGpio(FlPinName pinName, bool outputValue);
s32 (*const eat_WriteGpio)(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

4.4.7.4 ebdat6_27SetWatchDogGpio/eat_SetWatchDogGpio

The ebdat6_27SetWatchDogGpio/eat_SetWatchDogGpio function is used to configure the specific GPIO to kick the watch dog which is control by outside MCU. This function is only used when the module is updating the application.

- **Prototype**

```
s32 ebdat6_27SetWatchDogGpio(FlPinName gpio);
s32 (*const eat_SetWatchDogGpio)(FlPinName gpio);
```

- **Parameters**

gpio: The name of the GPIO pin which is used to be configured as kick-watchdog GPIO.

- **Return values**

FL_OK: Set GPIO successfully.
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/eat_KeySubscribe

The ebdat6_15KeySubscribe/eat_KeySubscribe function initializes the keypad pins to be keypad.

- **Prototype**

```
s32 ebdat6_15KeySubscribe(void);
s32 (*const eat_KeySubscribe)(void);
```

- **Return values**

FL_OK: Initialize successfully.

4.4.9 Periphery-I2C

Periphery-I2Cs are the I2C bus service pins. The I2C bus includes a SDA and a SCL pin.

I2C Name	Platform GPIOs	SIM900 Pin
SDA	GPIO39	SDA
SCL	GPIO38	SCL

4.4.9.1 ebdat15_01I2C_SpeedConfig/eat_I2C_SpeedConfig

The ebdat15_01I2C_SpeedConfig/eat_I2C_SpeedConfig function configures the speed of I2C bus.

- **Prototype**

```
void ebdat15_01I2C_SpeedConfig(I2C_SPEED_E i2c_speed);
void (*const eat_I2C_SpeedConfig)(I2C_SPEED_E i2c_speed);
```

- **Parameters**

i2c_speed: The speed of the I2C bus.

```
typedef enum
{
    I2C_SPEED_STD_RATE_100K       =0, /* 100 kHz */
    I2C_SPEED_FAST_RATE_400K      =1, /* 400 kHz */
    I2C_SPEED_HIGH_SPEED_3400K =2, /* 3.4 MHz (not support for SIM900 series)*/
}I2C_SPEED_E;
```

4.4.9.2 ebdat15_02I2C_ReadWriteDone/eat_I2C_ReadWriteDone

The ebdat15_02I2C_ReadWriteDone/eat_I2C_ReadWriteDone is a callback function.
After I2C operation finished, this function will be called once, and I2C bus status will be marked as idle status.

- **Prototype**

```
void ebdat15_02I2C_ReadWriteDone(GpsrTransferStatusType status, u32 userData);
void (*const eat_I2C_ReadWriteDone)(GpsrTransferStatusType status, u32 userData);
```

- **Parameters**

status:

GPSR_TRANSFER_DONE
GPSR_TRANSFER_FAILED

userData:

unsigned int type

4.4.9.3 ebdat15_03I2C_GetStatus/eat_I2C_GetStatus

The ebdat15_03I2C_GetStatus/eat_I2C_GetStatus function gets the operation status of I2C bus.

- **Prototype**

```
I2C_STATUS_E ebdat15_03I2C_GetStatus(void);
```

```
I2C_STATUS_E (*const eat_I2C_GetStatus)(void);
```

- **Return values**

```
I2C_STATUS_IDLE     /* idle */
I2C_STATUS_BUSY     /* busy */
```

4.4.9.4 ebdat15_04I2C_SetStatus/eat_I2C_SetStatus

The ebdat15_04I2C_SetStatus/eat_I2C_SetStatus function sets the operation status of I2C bus.

- **Prototype**

```
void ebdat15_04I2C_SetStatus(I2C_STATUS_E status);
void (*const eat_I2C_SetStatus)(I2C_STATUS_E status);
```

- **Parameters**

status:

```
I2C_STATUS_IDLE     /* idle */
I2C_STATUS_BUSY     /* busy */
```

4.4.9.5 ebdat15_05I2C_INITIALIZE_TRANSFER/eat_I2C_INITIALIZE_TRANSFER

The ebdat15_05I2C_INITIALIZE_TRANSFER/eat_I2C_INITIALIZE_TRANSFER function initializes the transfer of I2C bus. It returns the transfer parameter point. When it returns the point of GpsrTransferType, the context of this point should be initialized before ebdat15_06I2C_PUT_DATA or ebdat15_07I2C_GET_DATA is called.

- **Prototype**

```
GpsrTransferType*   ebdat15_05I2C_INITIALIZE_TRANSFER(void );
GpsrTransferType*   (*const eat_I2C_INITIALIZE_TRANSFER)(void);
```

- **Return values**

The point of the transfer parameter. About how to use this function, please refer to the example in the embeddedAT install package.

```
typedef struct GpsrTransfer
{
    void    *pCmd;         /* Pointer on command string */
    u32     cmdSize;       /* Size of the command string */
    void    *pTxData;      /* Pointer on data to be sent */
    u32     txDataSize;    /* Size of data to be sent */
    void    *pRxData;      /* Pointer on data to be received */
    u32     rxDataSize;    /* Size of data to be received */
```

```
    u32        userData;          /* General purpose user data */
    GpsrCallBackFunctionType *pIsrCallbackFct;        /* ISR callback function */
    struct GpsrTransfer          *pNext;     /* Reserved for the GPSR manager */
    GpsrAccessType                access;     /* Reserved for the GPSR manager */
}GpsrTransferType;
```

4.4.9.6 ebdat15_06I2C_PUT_DATA/eat_I2C_PUT_DATA

The ebdat15_06I2C_PUT_DATA/eat_I2C_PUT_DATA function puts data to I2C bus.

- **Prototype**

```
void ebdat15_06I2C_PUT_DATA(GpsrTransferType* PtRSFR);
void (*const eat_I2C_PUT_DATA)(GpsrTransferType* PtRSFR);
```

- **Parameters**

PtRSFR:

The data which you want to send.

Note: this parameter is got from the function ebdat15_05I2C_INITIALIZE_TRANSFER/eat_I2C_INITIALIZE_TRANSFER. So before this function is called, ebdat15_05I2C_INITIALIZE_TRANSFER/eat_I2C_INITIALIZE_TRANSFER must be called first. Please reference the I2C example to learn how to use it.

4.4.9.7 ebdat15_07I2C_GET_DATA/eat_I2C_GET_DATA

The ebdat15_07I2C _GET_DATA/eat_I2C_GET_DATA macro gets data from I2C bus.

- **Prototype**

```
void ebdat15_07I2C_GET_DATA(GpsrTransferType* PtRSFR);
void (*const eat_I2C_GET_DATA)(GpsrTransferType* PtRSFR);
```

- **Parameters**

PtRSFR:

The data which you want to receive.

Note: this parameter is got from the function ebdat15_05I2C_INITIALIZE_TRANSFER/eat_I2C_INITIALIZE_TRANSFER. So before this function is called, ebdat15_05I2C_INITIALIZE_TRANSFER/eat_I2C_INITIALIZE_TRANSFER must be called first. Please reference the I2C example to learn how to use it.

4.5 Audio API

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

4.5.1 ebdat10_01PlayContinousAudio/eat_PlayContinousAudio

The ebdat10_01PlayContinousAudio/eat_PlayContinousAudio function plays the continuous music in system.

- **Prototype**

```
bool ebdat10_01PlayContinousAudio(FlAudioName name);
bool (*const eat_PlayContinousAudio)(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/eat_StopContinousAudio

The ebdat10_02StopContinousAudio/eat_StopContinousAudio function stops playing continuous music

- **Prototype**

```
bool ebdat10_02StopContinousAudio(void) ;
bool (*const eat_StopContinousAudio)(void);
```

- **Return values**

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

4.5.3 ebdat10_03PlaySingleAudio/eat_PlaySingleAudio

The ebdat10_03PlaySingleAudio/eat_PlaySingleAudio 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) ;
bool (*const eat_PlaySingleAudio)(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/eat_PlaySingleAudioFromFile

The ebdat10_04PlaySingleAudioFromFile/eat_PlaySingleAudioFromFile function is used to play an audio file which is stored in the flash. It is played for local.
Note: When the amr file finished playing, "AMR_STOP" will report through event "MODEM_EVENT".

- **Prototype**

```
bool ebdat10_04PlaySingleAudioFromFile(u8* fileName);
bool (*const eat_PlaySingleAudioFromFile)(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 ebdat10_07PlayRemoteAmrFromFile/eat_PlayRemoteAmrFromFile

The ebdat10_07PlayRemoteAmrFromFile/eat_PlayRemoteAmrFromFile function is used to play an audio file which is stored in the flash. It is played for both remote and local.
Note: When the amr file finished playing, "AMR_STOP" will report through event "MODEM_EVENT".

- **Prototype**

```
bool    ebdat10_07PlayRemoteAmrFromFile(u8* fileName);
```

```
bool (*const eat_PlayRemoteAmrFromFile)(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.

4.5.6 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/eat_StartTimer

The ebdat8_01StartTimer/eat_StartTimer 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);
s32 (*const eat_StartTimer)(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/eat_StopTimer

The ebdat8_02StopTimer/eat_StopTimer function stops a Timer before it expires.

- **Prototype**

```
u16 ebdat8_02StopTimer(t_emb_Timer timer);
s32 (*const eat_StopTimer)(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/eat_SecondToTicks

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

- **Prototype**

```
u32 ebdat8_04SecondToTicks(u32 seconds);
u32 (*const eat_SecondToTicks)(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/eat_MillisecondToTicks

The ebdat8_05MillisecondToTicks/eat_MillisecondToTicks function converts time from milliseconds to KernelTicks.

- **Prototype**

```
u32 ebdat8_05MillisecondToTicks(u32 milliseconds);
u32 (*const eat_MillisecondToTicks)(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/eat_GetRelativeTime

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

- **Prototype**

```
s32    ebdat8_03GetRelativeTime(t_emb_Timer timer, u32 *tick) ;
s32    (*const eat_GetRelativeTime)(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 relative time successfully
FL_RET_ERR_PARAM: Incorrect parameter

4.6.7 ebdat8_06GetSystemTime/eat_GetSystemTime

The ebdat8_06GetSystemTime/eat_GetSystemTime function gets the local time.

- **Prototype**

```
s32 ebdat8_06GetSystemTime(t_emb_SysTimer * datetime);
s32 (*const eat_GetSystemTime)(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;
```

- **Return values**

FL_OK: Get the system time successfully
FL_RET_ERR_PARAM: Incorrect parameter

4.6.8 ebdat8_08GetSystemTickCounter/eat_GetSystemTickCounter

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

- **Prototype**

```
u32 ebdat8_08GetSystemTickCounter(void);
u32 (*const eat_GetSystemTickCounter)(void);
```

- **Return values**

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

4.6.9 ebdat8_10CurrentTaskSleep/eat_CurrentTaskSleep

The ebdat8_10CurrentTaskSleep/eat_CurrentTaskSleep function is used to make the app sleep. It is similar to the function ::Sleep in visual c++.

- **Prototype**

```
s32 ebdat8_10CurrentTaskSleep(u32 tick);
s32 (*const eat_CurrentTaskSleep)(u32 tick);
```

- **Prototype**

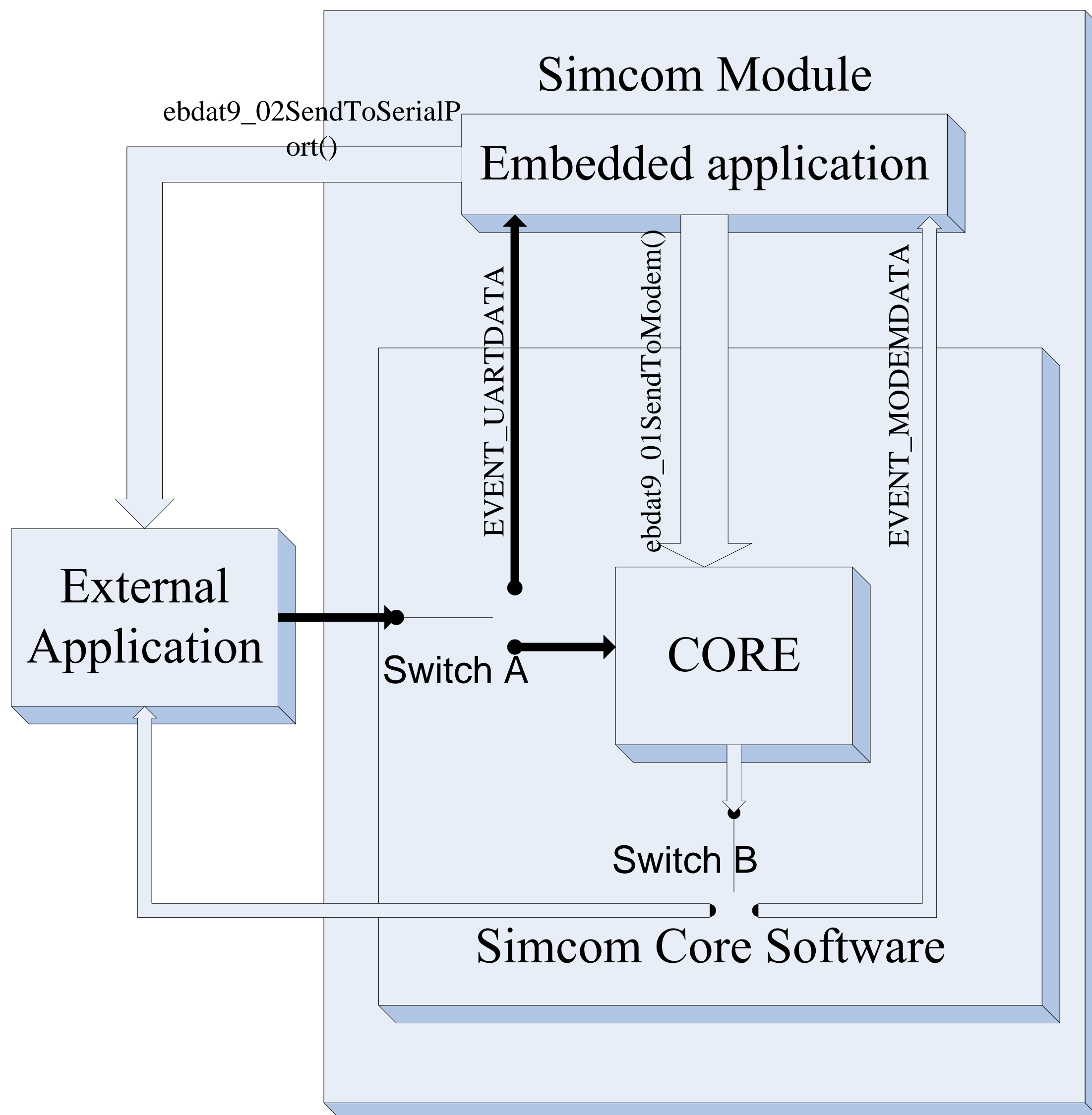
tick: the Kernel Ticks.

- **Return Value**

FL_OK: Sleep successfully
FL_RET_ERR_PARAM: Incorrect parameter

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/eat_SendToModem

The ebdat9_01SendToModem/eat_SendToModem 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);
s32 (*const eat_SendToModem)(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/eat_SendToSerialPort

The ebdat9_02SendToSerialPort/eat_SendToSerialPort 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);
s32 (*const eat_SendToSerialPort)(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/eat_SetModemdataToFL

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

- **Prototype**

```
void ebdat9_03SetModemdataToFL (bool destination);
void (*const eat_SetModemdataToFL)(bool destination);
```

- **Parameters**

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

4.7.4 ebdat9_04SetUartdataToFL/eat_SetUartdataToFL

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

- **Prototype**

```
void ebdat9_04SetUartdataToFL (bool destination);
void (*const eat_SetUartdataToFL)(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/eat_GetSerialPortTxStatus

The ebdat9_05GetSerialPortTxStatus/eat_GetSerialPortTxStatus 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);
```

```
bool (*const eat_GetSerialPortTxStatus)(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/eat_GetRTSPinLevel

The ebdat6_23GetRTSPinLevel/eat_GetRTSPinLevel 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);
u8 (*const eat_GetRTSPinLevel)(void);
```

- **Return values**

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

4.7.7 ebdat9_09ChangeMainUartBaudRate/eat_ChangeMainUartBaudRate

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

- **Prototype**

```
s32 ebdat9_09ChangeMainUartBaudRate(u32 BaudRate);
s32 (*const eat_ChangeMainUartBaudRate)(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/eat_GetMainUartBaudRate

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

- **Prototype**

```
u32 ebdat9_10GetMainUartBaudRate(void);
u32 (*const eat_GetMainUartBaudRate)(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/eat_ChangeMainUartDataFormat

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

- **Prototype**

```
s32   ebdat9_11ChangeMainUartDataFormat(FlMainUartDataFormat uartDataFormat);
s32 (*const eat_ChangeMainUartDataFormat)(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/eat_GetMainUartDataFormat

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

- **Prototype**

```
FlMainUartDataFormat ebdat9_12GetMainUartDataFormat(void);
FlMainUartDataFormat (*const eat_GetMainUartDataFormat)(void);
```

- **Return values**

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

4.7.11 ebdat9_13ChangeMainUartFlowControl/eat_ChangeMainUartFlowControl

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

- **Prototype**

```
s32 ebdat9_13ChangeMainUartFlowControl(FlMainUartFlowControlStruct flowControl);
s32 (*const eat_ChangeMainUartFlowControl)(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 the data format failed.

4.7.12 ebdat9_14GetMainUartFlowControl/eat_GetMainUartFlowControl

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

- **Prototype**

```
FlMainUartFlowControlStruct ebdat9_14GetMainUartFlowControl(void);
FlMainUartFlowControlStruct (*const eat_GetMainUartFlowControl)(void);
```

- **Return values**

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

4.7.13 ebdat9_15SubscribeURC/eat_SubscribeURC

The ebdat9_15SubscribeURC/eat_SubscribeURC 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);
s32 (*const eat_SubscribeURC)(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/eat_UnSubscribeURC

The ebdat9_16UnSubscribeURC/eat_UnSubscribeURC function is used to unsubscribe a URC.

- **Prototype**

```
s32 ebdat9_16UnSubscribeURC(u8 *urcString);
s32 (*const eat_UnSubscribeURC)(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/eat_GetURCNum

The ebdat9_17GetURCNum/eat_GetURCNum function is used to get the number of URCs which

have been subscribed.

- **Prototype**

```
u8 ebdat9_17GetURCNum(void);
u8 (*const eat_GetURCNum)(void);
```

- **Return values**

Return the number of URCs which have been subscribed.

4.7.16 ebdat9_19SubscribeATCommand/eat_SubscribeATCommand

The ebdat9_19SubscribeATCommand/eat_SubscribeATCommand 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);
s32 (*const eat_SubscribeATCommand)(ascii *urcString, u32 index);
```

- **Parameters**

urcString: The AT command will be unsubscribed. The maximum of the AT command which can be subscribed is 32. 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/eat_UnsubscribeATCommand

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

- **Prototype**

```
s32 ebdat9_20UnsubscribeATCommand(ascii *pString);
s32 (*const eat_UnsubscribeATCommand)(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.

4.7.18 ebdat9_24MainUartPortIsTransmitterEmpty/eat_MainUartPortIsTransmitterEmpty

The ebdat9_24MainUartPortIsTransmitterEmpty/eat_MainUartPortIsTransmitterEmpty function gets the transmit buffer's status of the serial port.

- **Prototype**

```
bool ebdat9_24MainUartPortIsTransmitterEmpty(void);
bool (*const eat_MainUartPortIsTransmitterEmpty)(void);
```

- **Return values**

TRUE: The transmit buffer is null.But note that the last byte is still being sent on the TX pin. You must wait the last byte sent for a fixed time.
FALSE: There are data in the transmit buffer

4.8 Debug API

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

4.8.1 ebdat7_00EnterDebugMode/eat_EnterDebugMode

The ebdat7_00EnterDebugMode/eat_EnterDebugMode 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);
void (*const eat_EnterDebugMode)(void);
```

4.8.2 ebdat7_01DebugTrace/eat_DebugTrace

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

- **Prototype**

```
void ebdat7_01DebugTrace (const u8 *Format, ... );
void (*const eat_DebugTrace) (const char *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/eat_DebugUartSend

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

- **Prototype**

```
s32 ebdat7_02DebugUartSend(u8 *buff, u16 len);
s32 (*const eat_DebugUartSend)(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 Other API

4.9.1 ebdat4_22GetADCValue/eat_GetADCValue

The ebdat4_22GetADCValue/eat_GetADCValue function is used to read ADC value from the core. It is the same as the second parameter of the AT command “AT+CADC?”

- **Prototype**

```
s32 ebdat4_22GetADCValue(u16 *voltage);
s32 (*const eat_GetADCValue)(u16 *voltage);
```

- **Parameters**

voltage: return the voltage of the ADC. Its unit is mV.

- **Return values**

FL_RET_ERR_PARAM: Parameter incorrect.
FL_OK: Get ADC value successfully.

4.9.2 ebdat4_23GetBatteryVoltage/eat_GetBatteryVoltage

The ebdat4_23GetBatteryVoltage/eat_GetBatteryVoltage function is used to read battery voltage from the core. It is the same as the AT command “AT+CBC”.

- **Prototype**

```
s32 ebdat4_23GetBatteryVoltage(CBCValue *voltage);
s32 (*const eat_GetBatteryVoltage)(CBCValue *voltage);
```

- **Parameters**

voltage: return the voltage of the battery voltage. Its unit is mV.

```
typedef struct CBCVALUETag
{
    u8 status;
    u8 percent;
    u16 voltage;
}CBCValue;
```

status: charge status.
0 ME is not charging.
1 ME is charging.

2 Charging has finished.

percent: Battery connection level. 1...100 battery has 1-100 percent of capacity remaining vent.

voltage: Battery voltage(mV)

- **Return values**

FL_RET_ERR_PARAM: Parameter incorrect.
FL_OK: Get battery voltage successfully.

4.9.3 ebdat4_24GetModuleTemperature/eat_GetModuleTemperature

The ebdat4_24GetModuleTemperature/eat_GetModuleTemperature function is used to get the temperature of the module from the core. It is the same as the second parameter of the AT command "AT+CMTE?"

- **Prototype**

```
s32 ebdat4_24GetModuleTemperature(s32 *tmp);
s32 (*const eat_GetModuleTemperature)(s32 *tmp);
```

- **Parameters**

tmp: return the temperature of the module. Its range is from -40 to 90.

- **Return values**

FL_RET_ERR_PARAM: Parameter incorrect.
FL_OK: Get temperature successfully.

4.9.4 ebdat4_25GetRegistrationStatus/eat_GetRegistrationStatus

The ebdat4_25GetRegistrationStatus/eat_GetRegistrationStatus function is used to get the network registration of the module from the core. It is the same as the second parameter of the AT command "AT+CREG?"

- **Prototype**

```
u8 ebdat4_25GetRegistrationStatus(void);
u8 (*const eat_GetRegistrationStatus)(void);
```

- **Return values**

0: Not registered, MT is not currently searching a new operator to register to
1: Registered, home network

2: Not registered, but MT is currently searching a new operator to register to
3: Registration denied
4: Unknown
5: Registered, roaming

4.9.5 ebdat4_26GetGPRSRegistrationStatus/eat_GetGPRSRegistrationStatus

The ebdat4_26GetGPRSRegistrationStatus/eat_GetGPRSRegistrationStatus function is used to get the GPRS registration status of the module from the core. It is the same as the second parameter of the AT command "AT+CGREG?"

- **Prototype**

```
u8 ebdat4_26GetGPRSRegistrationStatus(void);
u8 (*const eat_GetGPRSRegistrationStatus)(void);
```

- **Return values**

0: Not registered, MT is not currently searching an operator to register to.The GPRS service is disabled, the UE is allowed to attach for GPRS if requested by the user.
1: Registered, home network.
2: Not registered, but MT is currently trying to attach or searching an operator to register to. The GPRS service is enabled, but an allowable PLMN is currently not available. The UE will start a GPRS attach as soon as an allowable PLMN is available.
3: Registration denied. The GPRS service is disabled, the UE is not allowed to attach for GPRS if it is requested by the user.
4: Unknown
5: Registered, roaming

4.9.6 ebdat4_27GetGPRSAttachStatus/eat_GetGPRSAttachStatus

The ebdat4_27GetGPRSAttachStatus/eat_GetGPRSAttachStatus function is used to get the GPRS attach status of the module from the core. It is the same as the AT command "AT+CGATT?"

- **Prototype**

```
u8 ebdat4_27GetGPRSAttachStatus(void);
u8 (*const eat_GetGPRSAttachStatus)(void);
```

- **Return values**

0: The GPRS is detached
1: The GPRS is attached

4.9.7 ebdat4_28GetCSQValue/eat_GetCSQValue

The ebdat4_28GetCSQValue/eat_GetCSQValue function is used to get the signal quality report of the module from the core. It is the same as the AT command "AT+CSQ"

- **Prototype**

```
s32 ebdat4_28GetCSQValue(u8 *rssi, u8 *ber);
s32 (*const eat_GetCSQValue)(u8 *rssi, u8 *ber);
```

- **Parameters**

rssi: 0 115 dBm or less
1 111 dBm
2...30 110... 54 dBm
31 52 dBm or greater
99 not known or not detectable
ber: 0...7 As RXQUAL values in the table in GSM 05.08 [20] subclause 7.2.4
99 Not known or not detectable

- **Return values**

FL_RET_ERR_PARAM: Parameter incorrect.
FL_OK: Get signal quality report successfully.

4.9.8 ebdat4_29GetServiceCellInformation/eat_GetServiceCellInformation

The ebdat4_29GetServiceCellInformation/eat_GetServiceCellInformation function is used to get the service cell information of the module from the core.

- **Prototype**

```
ServiceCellInformation ebdat4_29GetServiceCellInformation(void);
ServiceCellInformation (*const eat_GetServiceCellInformation)(void);
```

- **Return values**

ServiceCellInformation:

```
typedef struct ServiceCellInformationTag
{
    u16 vl_Arfcn;       // Absolute radio frequency channel number
    u8 vl_RxLevel;      // Receive level
    u8 vl_Ber;          //Receive quality
    u16 vl_Mcc;         // Mobile country code
```

```
    u16 vl_Mnc;          // Mobile network code
    u8 vl_Bsic;          // Base station identity code
    u16 vl_CellID;       //Cell ID
    u8 vl_RxLevAccessMin;          // Receive level access minimum
    u8 vl_MsTxpwrMaxCch;        // Transmit power maximum CCCH
    u16 vl_Lac;              // Location area code
    u8 vl_TA;                // Timing Advance
}ServiceCellInformation;
```

4.9.9 ebdat4_30GetNeighborCellInformation/eat_GetNeighborCellInformation

The ebdat4_30GetNeighborCellInformation/eat_GetNeighborCellInformation function is used to get the neighbor cell information of the module from the core.

- **Prototype**

```
NeighborCellInfo ebdat4_30GetNeighborCellInformation(void);
NeighborCellInfo (*const eat_GetNeighborCellInformation)(void);
```

- **Return values**

NeighborCellInfo:

```
typedef struct NeighborCellInfoTag
{
    NeighborOneCellInfo cellInfo[6];
}NeighborCellInfo;
typedef struct NeighborOneCellInfoTag
{
    u16 vl_Arfcn;         // Absolute radio frequency channel number
    u8 vl_RxLevel;        // Receive level
    u8 vl_Bsic;           // Base station identity code
    u16 vl_CellID;        //Cell ID
    u16 vl_Mcc;           // Mobile country code
    u16 vl_Mnc;           // Mobile network code
    u16 vl_Lac;           // Location area code
}NeighborOneCellInfo;
```

4.9.10 ebdat4_31GetIMEI/eat_GetIMEI

The ebdat4_31GetIMEI/eat_GetIMEI function is used to get the IMEI of the module from the core.

- **Prototype**

```
IMEIValue ebdat4_31GetIMEI(void);
IMEIValue (*const eat_GetIMEI)(void);
```

- **Return values**

IMEIValue:

```
#define IMEI_VALUE_LENGTH                  15
typedef struct IMEIValueTag
{
    u8 imei[IMEI_VALUE_LENGTH + 1];
}IMEIValue;
```

4.9.11 ebdat4_32GetCfunValue/eat_GetCfunValue

The ebdat4_32GetCfunValue/eat_GetCfunValue function is used to get the CFUN value of the module from the core.

- **Prototype**

```
u8 ebdat4_32GetCfunValue(void);
u8 (*const eat_GetCfunValue)(void);
```

- **Return values**

The CFUN value of the module.

4.9.12 ebdat4_33GetModuleCpinStatus/eat_GetModuleCpinStatus

The ebdat4_33GetModuleCpinStatus/eat_GetModuleCpinStatus function is used to get the cpin value from the core. It is the same as the AT+CPIN?

- **Prototype**

```
s32 ebdat4_33GetModuleCpinStatus(u8 *value);
s32 (*const eat_GetModuleCpinStatus)(u8 *value);
```

- **Parameters**

***value:** return the value of the cpin.

```
enum
{
  EAT_READY,
```

```
    EAT_SIM_PIN,
    EAT_SIM_PUK,
    EAT_SIM_PIN2,
    EAT_SIM_PUK2,
    EAT_SIM_NCK,
    EAT_SIM_NSCK,
    EAT_SIM_SPCK,
    EAT_SIM_CCK,
    EAT_SIM_NO_PRESENT,
    EAT_MAX_CPIN_PARAM
};
```

- **Return values**

FL_RET_ERR_PARAM: Parameter incorrect.
FL_OK: Get cpin value successfully.

4.9.13 ebdat4_35GetSIMCardIMSI/eat_GetSIMCardIMSI

The ebdat4_35GetSIMCardIMSI/eat_GetSIMCardIMSI function is used to get the IMSI of the SIM card. It is the same as the AT command "AT+CIMI".

- **Prototype**

```
s32 ebdat4_35GetSIMCardIMSI(IMSIValue *imsi);
s32 (*const eat_GetSIMCardIMSI)(IMSIValue *imsi);
```

- **Parameters**

* **imsi:** return the IMSI of SIM card.

```
#define IMSI_VALUE_LENGTH                    15
typedef struct IMSIValueTag
{
    u8 imsi[IMSI_VALUE_LENGTH + 1];
}IMSIValue;
```

- **Return values**

FL_RET_ERR_PARAM: Parameter incorrect.
FL_OK: Get IMSI successfully.
FL_RET_ERR_SIM_NOT_INSERT: the SIM card is not inserted or the SIM card initialization has not been finished.
FL_RET_ERR_SIM_NOT_READY: IMSI has not been read from the SIM card. Please read it later.

4.9.14 ebdat4_36GetSIMCardICCID/eat_GetSIMCardICCID

The ebdat4_36GetSIMCardICCID/eat_GetSIMCardICCID function is used to get the ICCID of the SIM card. It is the same as "AT+CCID".

- **Prototype**

```
s32 ebdat4_36GetSIMCardICCID(ICCIDValue *iccid);
s32 (*const eat_GetSIMCardICCID)(ICCIDValue *iccid);
```

- **Parameters**

*** iccid:** return the ICCID of SIM card.

- **Return values**

FL_RET_ERR_PARAM: Parameter incorrect.
FL_OK: Get ICCID successfully.
FL_RET_ERR_SIM_NOT_INSERT: the SIM card is not inserted or the SIM card initialization has not been finished.

4.9.15 ebdat4_37GetSIMCardSPN/eat_GetSIMCardSPN

The ebdat4_37GetSIMCardSPN/eat_GetSIMCardSPN function is used to get the service provider name from the SIM card. It is the same as "AT+CSPN?".

- **Prototype**

```
s32 ebdat4_37GetSIMCardSPN(SPNValue *spn);
s32 (*const eat_GetSIMCardSPN)(SPNValue *spn);
```

- **Parameters**

*** iccid:** return the ICCID of SIM card.

- **Return values**

FL_RET_ERR_PARAM: Parameter incorrect.
FL_OK: Get ICCID successfully.
FL_RET_ERR_SIM_NOT_INSERT: the SIM card is not inserted or the SIM card initialization has not been finished.

4.9.16 ebdat4_38SetSMSIndEvent/eat_SetSMSIndEvent

The ebdat4_38SetSMSIndEvent/eat_SetSMSIndEvent function is used to enable short message

indication when the module receive a short message.

- **Prototype**

```
void ebdat4_38SetSMSIndEvent(bool enSmsInd);
void (*const eat_SetSMSIndEvent)(bool enSmsInd);
```

- **Parameters**

enSmsInd: enable SMS indicate.

4.9.17 ebdat4_39SetCregIndEvent/eat_SetCregIndEvent

The ebdat4_39SetCregIndEvent/eat_SetCregIndEvent function is used to enable CREG indication when the CREG value is changed.

- **Prototype**

```
void ebdat4_39SetCregIndEvent(bool enCreg);
void (*const eat_SetCregIndEvent)(bool enCreg);
```

- **Parameters**

enCreg: enable CREG indication.

4.9.18 ebdat4_40SetCgregIndEvent/eat_SetCgregIndEvent

The ebdat4_40SetCgregIndEvent/eat_SetCgregIndEvent function is used to enable CGREG indication when the CGREG value is changed.

- **Prototype**

```
void ebdat4_40SetCgregIndEvent(bool enCgreg);
void (*const eat_SetCgregIndEvent)(bool enCgreg);
```

- **Parameters**

enCgreg: enable CGREG indication.

4.9.19 ebdat4_34GetCurrentTaskID/eat_GetCurrentTaskID

The ebdat4_34GetCurrentTaskID/eat_GetCurrentTaskID function gets the current task ID.

- **Prototype**

```
u8 ebdat4_34GetCurrentTaskID(void);
u8 (*const eat_GetCurrentTaskID)(void);
```

- **Return values**

Return the task of the current running task. As following:
FL_EAT_TASK, FL_MULTI_TASK_1, FL_MULTI_TASK_2, FL_MULTI_TASK_3, FL_MULTI_TASK_4, FL_MULTI_TASK_5

4.10 Standard library API

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

4.10.1 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.

4.10.2 ebdat4_10strRemoveCRLF/eat_strRemoveCRLF

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

- **Prototype**

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

- **Parameters**

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

- **Return values**

Modified string

4.10.3 ebdat4_11strGetParameterString/eat_strGetParameterString

The ebdat4_11strGetParameterString/eat_strGetParameterString function returns parameter string at a given position

- **Prototype**

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

- **Parameters**

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

- **Return values**

Address to the parameter string

4.10.4 ebdat6_17DisablePowerOffKey/eat_DisablePowerOffKey

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

- **Prototype**

```
void   ebdat6_17DisablePowerOffKey ( void );
void (*const eat_DisablePowerOffKey)(void);
```

4.10.5 ebdat6_18EnablePowerOffKey/eat_EnablePowerOffKey

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

- **Prototype**

```
void   ebdat6_18EnablePowerOffKey ( void );
void (*const eat_EnablePowerOffKey)(void);
```

4.10.6 ebdat4_15ExitOutOfSleepMode/eat_ExitOutOfSleepMode

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

- **Prototype**

```
s32 ebdat4_15ExitOutOfSleepMode(void);
s32 (*const eat_ExitOutOfSleepMode)(void);
```

- **Return values**

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

4.10.7 ebdat4_17EnterSleepMode/eat_EnterSleepMode

The ebdat4_17EnterSleepMode/eat_EnterSleepMode 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);
s32 (*const eat_EnterSleepMode)(void);
```

- **Return values**

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

4.11 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.

4.11.1 ebdat11_10GprsActive/eat_GprsActive

The ebdat11_10GprsActive/eat_GprsActive function is used to activate gprs bearer.

- **Prototype**

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

4.11.2 ebdat11_15GprsDeactive/eat_GprsDeactive

The ebdat11_15GprsDeactive/eat_GprsDeactive function is used to release gprs bearer.

- **Prototype**

```
s32 ebdat11_15GprsDeactive(void);
s32 (*const eat_GprsDeactive)(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.

4.11.3 ebdat11_20SocketConnect/eat_SocketConnect

The ebdat11_20SocketConnect/eat_SocketConnect function sets up TCP andUDP socket.

- **Prototype**

```
u32 ebdat11_20SocketConnect(FlSocketType_e type,u8 *url, u16 sockPort);
u32 (*const eat_SocketConnect)(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 0XFFFFFFFF, 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.

4.11.4 ebdat11_25SocketClose/eat_SocketClose

The ebdat11_25SocketClose/eat_SocketClose function is used to close the socket.

- **Prototype**

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

```
s32 (*const eat_SocketClose)(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.

4.11.5 ebdat11_30SocketSend/eat_SocketSend

The ebdat11_30SocketSend/eat_SocketSend function sends socket data.

- **Prototype**

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

4.11.6 ebdat11_35SocketRecv/eat_SocketRecv

The ebdat11_35SocketRecv/eat_SocketRecv function is used to read socket data.

- **Prototype**

```
u16 ebdat11_35SocketRecv(u32 socket,void *buf_p, u16 len,u16 *remain);
u16 (*const eat_SocketRecv)(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.

4.11.7 ebdat11_45SocketTcpServerSet/eat_SocketTcpServerSet

The ebdat11_45SocketTcpServerSet/eat_SocketTcpServerSet function sets up and close tcp server.

- **Prototype**

```
s32 ebdat11_45SocketTcpServerSet(u8 mode,u16 port);
```

```
s32 (*const eat_SocketTcpServerSet)(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.

4.11.8 ebdat11_50GetLocalIpAddr/eat_GetLocalIpAddr

The ebdat11_50GetLocalIpAddr/eat_GetLocalIpAddr function gets local IP address.

- **Prototype**

```
u32 ebdat11_50GetLocalIpAddr(void);
u32 (*const eat_GetLocalIpAddr)(void);
```

- **Return values**

Return the local IP address of ebdat GPRS bearer.

4.12 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

4.13 Updating Embedded Application/eat_UpdateEmbeddedAp

The eat1_09UpdateEmbeddedAp/eat_UpdateEmbeddedAp function initiates the embedded application updating procedure.

- **Prototype**

```
s32 eat1_09UpdateEmbeddedAp( u16 startID, u16 idCount, u32 osSize) ;
s32 (*const eat_UpdateEmbeddedAp)( 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 for the 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.

4.14 DTMF API

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

Note: This event is only existed in DTMF firmware. It is not supported in normal version.

4.14.1 ebdat10_06DTMFDetectEnable/eat_DTMFDetectEnable

The ebdat10_06DTMFDetectEnable/eat_DTMFDetectEnable function is used to enable/disable DTMF detect function.

- **Prototype**

```
s32 ebdat10_06DTMFDetectEnable (bool isEnable);
s32 (*const eat_DTMFDetectEnable)(bool isEnable);
```

- **Parameter**

isEnable: 0 disable
1 enable

- **Return values**

FL_OK: DTMF detection set successfully
FL_ERROR: Incorrect parameter

4.15 SIM card API

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

4.15.1 ebdat13_00SetModemAPDUToFL/eat_SetModemAPDUToFL

The ebdat13_00SetModemAPDUToFL/eat_SetModemAPDUToFL function is used to control the direction of the APDU data from the core.

- **Prototype**

```
void ebdat13_00SetModemAPDUToFL(bool destination);
void (*const eat_SetModemAPDUToFL)(bool destination);
```

- **Parameter**

destination: **TRUE:** Send the APDU data from core to embedded application.
FALSE: It is directed to the SIM card.

4.15.2 ebdat13_01SetSIMCardAPDUToFL/eat_SetSIMCardAPDUToFL

The ebdat13_01SetSIMCardAPDUToFL/eat_SetSIMCardAPDUToFL function is used to control the direction of the APDU data from the SIM card.

- **Prototype**

```
void ebdat13_01SetSIMCardAPDUToFL(bool destination);
void (*const eat_SetSIMCardAPDUToFL)(bool destination);
```

- **Parameter**

destination: **TRUE:** Send the APDU data from the SIM card to the embedded application.
FALSE: It is directed to the core.

4.15.3 ebdat13_03SendResetReqToSIMCard/eat_SendResetReqToSIMCard

The ebdat13_03SendResetReqToSIMCard/eat_SendResetReqToSIMCard function is used to send the reset request to the SIM card.

- **Prototype**

```
void ebdat13_03SendResetReqToSIMCard(u8 type);
void (*const eat_SendResetReqToSIMCard)(u8 type);
```

- **Parameter**

type: the reset type

4.15.4 ebdat13_05SendSIMCardResetCnfToModem/eat_SendSIMCardResetCnfToModem

The ebdat13_05SendSIMCardResetCnfToModem/eat_SendSIMCardResetCnfToModem function is used to send the reset confirmation to the Modem. Note if soft SIM is used, that means there is no real SIM card connected to the module, when a MODEMAPDU_EVT is received and its type is **FL_MOD_APDU_RESET**, the app should use this function to respond to the core.

- **Prototype**

```
void ebdat13_05SendSIMCardResetCnfToModem(SIMCARDRESET_CNF      resetCnf);
void (*const eat_SendSIMCardResetCnfToModem)(SIMCARDRESET_CNF      resetCnf);
```

- **Parameter**

resetCnf: The reset confirmation information. See character 3.2.12

4.15.5 ebdat13_08SendAPDUReqToSIMCard/eat_SendAPDUReqToSIMCard

The ebdat13_08SendAPDUReqToSIMCard/eat_SendAPDUReqToSIMCard function is used to send APDU data to SIM card.

- **Prototype**

```
s32 ebdat13_08SendAPDUReqToSIMCard(MODEMAPDU_DATA apduData, u8 type);
```

```
s32 (*const eat_SendAPDUReqToSIMCard)(MODEMAPDU_DATA apduData, u8 type);
```

- **Parameter**

apduData: The APDU data which is sent to the SIM card. See character 3.2.11
type: 0: if no data is in **apduData.a_CData**, the type should be 0.
1: if there are some data in **apduData.a_CData**, the type should be 1.

- **Return values**

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

4.15.6 ebdat13_10SendAPDUCnfToModem/eat_SendAPDUCnfToModem

The ebdat13_10SendAPDUCnfToModem/eat_SendAPDUCnfToModem function is used to send the APDU data which is received from the real SIM card back to the core.
Note: If soft SIM is used, this function should not be called. This function is only used to send the data which is from the real SIM card back to the core.

- **Prototype**

```
void ebdat13_10SendAPDUCnfToModem(SIMCARDAPDU_DATA apduData);
void (*const eat_SendAPDUCnfToModem)(SIMCARDAPDU_DATA apduData);
```

- **Parameter**

apduData: The APDU data which is sent to the core. See character 3.2.12

4.15.7 ebdat13_11SoftSendAPDUCnfToModem/eat_SoftSendAPDUCnfToModem

The ebdat13_11SoftSendAPDUCnfToModem/eat_SoftSendAPDUCnfToModem function is used to send the APDU data to the core directly, when the EVENT_MODEM_APDU event is triggered and its apduType is FL_MOD_APDU_REQ_DATA or FL_MOD_APDU_SEND_DATA.
Note: If soft SIM is used, this function should be called instead of ebdat13_10SendAPDUCnfToModem.

- **Prototype**

```
void  ebdat13_11SoftSendAPDUCnfToModem(SIMCARDAPDU_DATA apduData);
void (*const eat_SoftSendAPDUCnfToModem)(SIMCARDAPDU_DATA apduData);
```

- **Parameter**

apduData: The APDU data which is sent to the core. See character 3.2.12

4.16 Multi task API

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

4.16.1 ebdat4_21SendEventMsg/eat_SendEventMsg

The ebdat4_21SendEventMsg/eat_SendEventMsg function is used to send the message to other tasks or the current task.

- **Prototype**

```
s32 ebdat4_21SendEventMsg(MSG_EVT msg);
s32 (*const eat_SendEventMsg)(MSG_EVT msg);
```

- **Parameter**

msg: The message which is sent to other tasks or the current task.
Note: When a message is wanted to be received, eat1_02GetEvent should be called. And then MSG_EVT event will be received.

- **Return values**

FL_OK: Send message successfully.
FL_RET_ERR_PARAM: Parameter incorrect.

4.16.2 ebdat14_00CreateSem/eat_CreateSem

The ebdat14_00CreateSem/eat_CreateSem function is used to initialize a semaphore.

- **Prototype**

```
s32 ebdat14_00CreateSem(FlSemaphoreID sem, int vp_Count, int vp_CountMax);
s32 (*const eat_CreateSem)(FlSemaphoreID sem, int vp_Count, int vp_CountMax);
```

- **Parameter**

sem: The semaphore ID which is wanted to be created.

```
FlSemaphoreID
  typedef enum FlSemaphoreIDTag
  {
  FL_SEM_0,
```

```
        FL_SEM_1,
        FL_SEM_2,
        FL_SEM_3,
        FL_SEM_4,
        FL_SEM_5,
        NUM_OF_SEM
       }FlSemaphoreID;
```

vp_Count: The default value of the semaphore.
vp_CountMax: The maximum of the semaphore. It cannot be set over 10.

- **Return values**

FL_OK: Create the semaphore successfully.
FL_RET_ERR_PARAM: Parameter incorrect.

4.16.3 ebdat14_01semPend/eat_semPend

The ebdat14_01semPend/eat_semPend function is used to obtain an instance of the specified semaphore.

- **Prototype**

```
s32 ebdat14_01semPend(FlSemaphoreID sem);
s32 (*const eat_semPend)(FlSemaphoreID sem);
```

- **Parameter**

sem: The semaphore ID which is wanted to be obtained.

- **Return values**

FL_OK: Obtain the semaphore successfully.
FL_RET_ERR_PARAM: Parameter incorrect.

4.16.4 ebdat14_02semPost/eat_semPost

The ebdat14_02semPost/eat_semPost function is used to release an instance of the specified semaphore.

- **Prototype**

```
s32 ebdat14_02semPost(FlSemaphoreID sem);
s32 (*const eat_semPost)(FlSemaphoreID sem);
```

- **Parameter**

sem: The semaphore ID which is wanted to be released.

- **Return values**

FL_OK: Release the semaphore successfully.
FL_RET_ERR_PARAM: Parameter incorrect.

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