Jennic

TECHNOLOGY FOR A CHANGING WORLD

Integrated Peripherals API Reference Manual

JN-RM-2001

Revision 3.2

16 July 2010

Contents

About this Manual 11

Organisation 11

Conventions 12

Acronyms and Abbreviations 13

Related Documents 13

Feedback Address 14

1. Overview 15

2. General Functions 17

u32AHI_Init 18

bAHI_PhyRadioSetPower (JN5139/JN5148 Only) 19

vAppApiSetBoostMode (JN5139 Only) 20

vAHI_HighPowerModuleEnable 21

vAHI_ETSIHighPowerModuleEnable (JN5148 Only) 22

vAHI_AntennaDiversityOutputEnable 23

vAHI_BbcSetHigherDataRate (JN5148 Only) 24

vAHI_BbcSetInterFrameGap (JN5148 Only) 25

vAHI_StartRandomNumberGenerator (JN5148 Only) 26

vAHI_StopRandomNumberGenerator (JN5148 Only) 27

u16AHI_ReadRandomNumber (JN5148 Only) 28

bAHI_RndNumPoll (JN5148 Only) 29

vAHI_SetStackOverflow (JN5148 Only) 30

3. System Controller 31

u8AHI_PowerStatus 34

vAHI_MemoryHold 35

vAHI_CpuDoze 36

vAHI_PowerDown (JN5121 Only) 37

vAHI_Sleep (JN5139/JN5148 Only) 38

vAHI_ProtocolPower 40

vAHI_ExternalClockEnable (JN5139 Only) 41

bAHI_Set32KhzClockMode (JN5148 Only) 42

vAHI_SelectClockSource (JN5148 Only) 43

bAHI_GetClkSource (JN5148 Only) 44

bAHI_SetClockRate (JN5148 Only) 45

u8AHI_GetSystemClkRate (JN5148 Only) 46

vAHI_EnableFastStartUp (JN5148 Only) 47

vAHI_PowerXTAL (JN5148 Only) 48

vAHI_BrownOutConfigure (JN5148 Only) 49

bAHI_BrownOutStatus (JN5148 Only) 51

bAHI_BrownOutEventResetStatus (JN5148 Only) 52
u32AHI_BrownOutPoll (JN5148 Only) 53
vAHI_SwReset 54
vAHI_DriveResetOut 55
vAHI_ClearSystemEventStatus 56
vAHI_SysCtrlRegisterCallback 57

4. Analogue Peripherals 59

4.1 Common Analogue Peripheral Functions 59
vAHI_ApConfigure 60
vAHI_ApSetBandGap 61
bAHI_APRegulatorEnabled 62
vAHI_APRegisterCallback 63

4.2 ADC Functions 64
vAHI_AdcEnable 65
vAHI_AdcStartSample 66
vAHI_AdcStartAccumulateSamples (JN5148 Only) 67
bAHI_AdcPoll (JN5139/JN5148 Only) 68
u16AHI_AdcRead 69
vAHI_AdcDisable 70

4.3 DAC Functions 71
vAHI_DacEnable 72
vAHI_DacOutput 73
bAHI_DacPoll 74
vAHI_DacDisable 75

4.4 Comparator Functions 76
vAHI_CompEnable (JN5121 Only) 77
vAHI_CompDisable (JN5121 Only) 78
vAHI_CompIntEnable (JN5121 Only) 79
u8AHI_CompStatus (JN5121 Only) 80
u8AHI_CompWakeStatus (JN5121 Only) 81
vAHI_ComparatorEnable (JN5139/JN5148 Only) 82
vAHI_ComparatorDisable (JN5139/JN5148 Only) 83
vAHI_ComparatorLowPowerMode (JN5139/JN5148 Only) 84
vAHI_ComparatorIntEnable (JN5139/JN5148 Only) 85
u8AHI_ComparatorStatus (JN5139/JN5148 Only) 86
u8AHI_ComparatorWakeStatus (JN5139/JN5148 Only) 87

5. DIOs 89

vAHI_DioSetDirection 90
vAHI_DioSetOutput 91
u32AHI_DioReadInput 92
vAHI_DioSetPullup 93
vAHI_DioSetByte (JN5148 Only) 94
u8AHI_DioReadByte (JN5148 Only) 95

vAHI_DioInterruptEnable 96
vAHI_DioInterruptEdge 97
u32AHI_DioInterruptStatus 98
vAHI_DioWakeEnable 99
vAHI_DioWakeEdge 100
u32AHI_DioWakeStatus 101

6. UARTs 103

vAHI_UartEnable 104
vAHI_UartDisable 105
vAHI_UartSetBaudRate 106
vAHI_UartSetBaudDivisor 107
vAHI_UartSetClocksPerBit (JN5148 Only) 108
vAHI_UartSetControl 109
vAHI_UartSetInterrupt 110
vAHI_UartSetRTSCTS (JN5139/JN5148 Only) 111
vAHI_UartSetRTS (JN5148 Only) 112
vAHI_UartSetAutoFlowCtrl (JN5148 Only) 113
vAHI_UartSetBreak (JN5148 Only) 115
vAHI_UartReset 116
u8AHI_UartReadRxFifoLevel (JN5148 Only) 117
u8AHI_UartReadTxFifoLevel (JN5148 Only) 118
u8AHI_UartReadLineStatus 119
u8AHI_UartReadModemStatus 120
u8AHI_UartReadInterruptStatus 121
vAHI_UartWriteData 122
u8AHI_UartReadData 123
vAHI_Uart0RegisterCallback 124
vAHI_Uart1RegisterCallback 125

7. Timers 127

vAHI_TimerEnable 130
vAHI_TimerClockSelect (JN5148 Only) 132
vAHI_TimerConfigureOutputs (JN5148 Only) 133
vAHI_TimerConfigureInputs (JN5148 Only) 134
vAHI_TimerStartSingleShot 135
vAHI_TimerStartRepeat 136
vAHI_TimerStartCapture 137
vAHI_TimerStartDeltaSigma 138
u16AHI_TimerReadCount 140
vAHI_TimerReadCapture 141
vAHI_TimerReadCaptureFreeRunning 142
vAHI_TimerStop 143
vAHI_TimerDisable 144
vAHI_TimerDIOControl (JN5139/JN5148 Only) 145
vAHI_TimerFineGrainDIOControl (JN5148 Only) 146

u8AHI_TimerFired 147
vAHI_Timer0RegisterCallback 148
vAHI_Timer1RegisterCallback 149
vAHI_Timer2RegisterCallback (JN5148 Only) 150

8. Wake Timers 151

vAHI_WakeTimerEnable 152
vAHI_WakeTimerStart (JN5139/JN5121 Only) 153
vAHI_WakeTimerStartLarge (JN5148 Only) 154
vAHI_WakeTimerStop 155
u32AHI_WakeTimerRead (JN5139/JN5121 Only) 156
u64AHI_WakeTimerReadLarge (JN5148 Only) 157
u8AHI_WakeTimerStatus 158
u8AHI_WakeTimerFiredStatus 159
u32AHI_WakeTimerCalibrate 160

9. Tick Timer 161

vAHI_TickTimerConfigure 162
vAHI_TickTimerInterval 163
vAHI_TickTimerWrite 164
u32AHI_TickTimerRead 165
vAHI_TickTimerIntEnable 166
bAHI_TickTimerIntStatus 167
vAHI_TickTimerIntPendClr 168
vAHI_TickTimerInit (JN5139/JN5121 Only) 169
vAHI_TickTimerRegisterCallback (JN5148 Only) 170

10. Watchdog Timer (JN5148 Only) 171

vAHI_WatchdogStart (JN5148 Only) 172
vAHI_WatchdogStop (JN5148 Only) 173
vAHI_WatchdogRestart (JN5148 Only) 174
u16AHI_WatchdogReadValue (JN5148 Only) 175
bAHI_WatchdogResetEvent (JN5148 Only) 176

11. Pulse Counters (JN5148 Only) 177

bAHI_PulseCounterConfigure (JN5148 Only) 178
bAHI_SetPulseCounterRef (JN5148 Only) 180
bAHI_StartPulseCounter (JN5148 Only) 181
bAHI_StopPulseCounter (JN5148 Only) 182
u32AHI_PulseCounterStatus (JN5148 Only) 183
bAHI_Read16BitCounter (JN5148 Only) 184
bAHI_Read32BitCounter (JN5148 Only) 185
bAHI_Clear16BitPulseCounter (JN5148 Only) 186
bAHI_Clear32BitPulseCounter (JN5148 Only) 187

12. Serial Interface (2-wire) 189

12.1SI Master Functions 190

vAHI_SiConfigure (JN5139 Only) 191
vAHI_SiMasterConfigure (JN5148 Only) 192
vAHI_SiMasterDisable (JN5148 Only) 193
bAHI_SiMasterSetCmdReg 194
vAHI_SiMasterWriteSlaveAddr 196
vAHI_SiMasterWriteData8 197
u8AHI_SiMasterReadData8 198
bAHI_SiMasterPollBusy 199
bAHI_SiMasterPollTransferInProgress 200
bAHI_SiMasterCheckRxNack 201
bAHI_SiMasterPollArbitrationLost 202
vAHI_SiRegisterCallback 203

12.2SI Slave Functions (JN5148 Only) 204

vAHI_SiSlaveConfigure (JN5148 Only) 205
vAHI_SiSlaveDisable (JN5148 Only) 207
vAHI_SiSlaveWriteData8 (JN5148 Only) 208
u8AHI_SiSlaveReadData8 (JN5148 Only) 209
vAHI_SiRegisterCallback 210

13. Serial Peripheral Interface (SPI Master) 211

vAHI_SpiConfigure (JN5121 Version) 212
vAHI_SpiConfigure (JN5139/JN5148 Version) 214
vAHI_SpiReadConfiguration 216
vAHI_SpiRestoreConfiguration 217
vAHI_SpiSelect 218
vAHI_SpiStop 219
vAHI_SpiStartTransfer (JN5148 Only) 220
vAHI_SpiStartTransfer32 (JN5139/JN5121 Only) 221
u32AHI_SpiReadTransfer32 222
vAHI_SpiStartTransfer16 (JN5139/JN5121 Only) 223
u16AHI_SpiReadTransfer16 224
vAHI_SpiStartTransfer8 (JN5139/JN5121 Only) 225
u8AHI_SpiReadTransfer8 226
vAHI_SpiContinuous (JN5148 Only) 227
bAHI_SpiPollBusy 228
vAHI_SpiWaitBusy 229
vAHI_SetDelayReadEdge (JN5148 Only) 230
vAHI_SpiRegisterCallback 231

14. Intelligent Peripheral Interface (SPI Slave) 233

vAHI_IpEnable (JN5148 Version) 234
vAHI_IpEnable (JN5139/JN5121 Version) 235
vAHI_IpDisable (JN5148 Only) 236

bAHI_IpSendData (JN5148 Version) 237
bAHI_IpSendData (JN5139/JN5121 Version) 238
bAHI_IpReadData (JN5148 Version) 239
bAHI_IpReadData (JN5139/JN5121 Version) 240
bAHI_IpTxDone 241
bAHI_IpRxDataAvailable 242
vAHI_IpReadyToReceive (JN5148 Only) 243
vAHI_IpRegisterCallback 244

15. Digital Audio Interface (JN5148 Only) 245

vAHI_DaiEnable (JN5148 Only) 246
vAHI_DaiSetBitClock (JN5148 Only) 247
vAHI_DaiSetAudioData (JN5148 Only) 248
vAHI_DaiSetAudioFormat (JN5148 Only) 249
vAHI_DaiConnectToFIFO (JN5148 Only) 250
vAHI_DaiWriteAudioData (JN5148 Only) 251
vAHI_DaiReadAudioData (JN5148 Only) 252
vAHI_DaiStartTransaction (JN5148 Only) 253
bAHI_DaiPollBusy (JN5148 Only) 254
vAHI_DaiInterruptEnable (JN5148 Only) 255
vAHI_DaiRegisterCallback (JN5148 Only) 256

16. Sample FIFO Interface (JN5148 Only) 257

vAHI_FifoEnable (JN5148 Only) 258
bAHI_FifoRead (JN5148 Only) 259
vAHI_FifoWrite (JN5148 Only) 260
u8AHI_FifoReadRxLevel (JN5148 Only) 261
u8AHI_FifoReadTxLevel (JN5148 Only) 262
vAHI_FifoSetInterruptLevel (JN5148 Only) 263
vAHI_FifoEnableInterrupts (JN5148 Only) 264
vAHI_FifoRegisterCallback (JN5148 Only) 265

17. External Flash Memory 267

bAHI_FlashInit 268
bAHI_FlashErase (JN5121/JN5139 Only) 269
bAHI_FlashEraseSector 270
bAHI_FlashProgram (JN5121/JN5139 Only) 271
bAHI_FullFlashProgram 272
bAHI_FlashRead (JN5121/JN5139 Only) 273
bAHI_FullFlashRead 274
vAHI_FlashPowerDown (JN5139/JN5148 Only) 275
vAHI_FlashPowerUp (JN5139/JN5148 Only) 276

Appendices 277

A. Interrupt Handling 277

A.1 Callback Function Prototype and Parameters 277

A.2 Peripheral Interrupt Enumerations (u32DeviceId) 278

A.3 Peripheral Interrupt Sources (u32ItemBitmap) 279

About this Manual

This manual details the C functions of the Jennic Integrated Peripherals Application Programming Interface (API), which allows a wireless network application to interact with the peripherals on a Jennic wireless microcontroller. These functions can be used to set up, control and respond to the on-chip peripheral blocks, such as UARTs, timers and general-purpose digital IO lines, amongst others.

Note 1: This manual covers the full range of Jennic wireless microcontrollers. Where a function or set of functions is applicable only to a specific microcontroller, this is clearly indicated (e.g. “JN5148 Only”).

Note 2: You should refer to this manual in conjunction with the Jennic *Integrated Peripherals API User Guide (JN-UG-3066)*, which provides guidance on using the API functions to control the JN5148/JN5139 on-chip peripherals.

Organisation

This manual consists of 17 chapters and an appendix, as follows:

- Chapter 1 presents a functional overview of the Integrated Peripherals API.
- Chapter 2 details the **General functions** of the API, including the API initialisation function.
- Chapter 3 details the **System Controller functions**, including functions that configure the system clock and sleep operations.
- Chapter 4 details the **Analogue Peripheral functions**, used to control the ADC, DACs and comparators.
- Chapter 5 details the **DIO functions**, used to control the 21 general-purpose digital input/output pins.
- Chapter 6 details the **UART functions**, used to control the two 16550-compatible UARTs.
- Chapter 7 details the **Timer functions**, used to control the general-purpose. timers.
- Chapter 8 details the **Wake Timer functions**, used to control the wake timers that can be employed to time sleep periods.
- Chapter 9 details the **Tick Timer functions**, used to control the high-precision hardware timer.
- Chapter 10 details the **Watchdog Timer functions (JN5148 only)**, used to control the watchdog that allows software lock-ups to be avoided.
- Chapter 11 details the **Pulse Counter functions (JN5148 only)**, used to control the two pulse counters.

- Chapter 12 details the **Serial Interface (SI) functions**, used to control a 2-wire SI master (all chips) and SI slave (JN5148 only).
- Chapter 13 details the **Serial Peripheral Interface (SPI) functions**, used to control the master interface to the SPI bus.
- Chapter 14 details the **Intelligent Peripheral (IP) Interface functions**, used to control the IP interface (acts as a SPI slave).
- Chapter 15 details the **Digital Audio Interface (DAI) functions (JN5148 only)**, used to control the interface to an external audio device.
- Chapter 16 details the **Sample FIFO Interface functions (JN5148 only)**, used to control the optional FIFO buffer between the CPU and the DAI.
- Chapter 17 details the **Flash Memory functions**, used to manage the external Flash memory.
- The Appendices detail resources used in the handling of peripheral device interrupts.

Conventions

Files, folders, functions and parameter types are represented in **bold** type.

Function parameters are represented in *italics* type.

Code fragments are represented in the `Courier New` typeface.

💡	This is a **Tip**. It indicates useful or practical information.

ℹ	This is a **Note**. It highlights important additional information.

⚠	*This is a **Caution**. It warns of situations that may result in equipment malfunction or damage.*

Acronyms and Abbreviations

ADC	Analogue-to-Digital Converter
AES	Advanced Encryption Standard
AHI	Application Hardware Interface
API	Application Programming Interface
CPU	Central Processing Unit
CTS	Clear-To-Send
DAC	Digital-to-Analogue Converter
DAI	Digital Audio Interface
DIO	Digital Input/Output
EIRP	Equivalent Isotropically Radiated Power
FIFO	First In, First Out (queue)
IP	Intelligent Peripheral
MAC	Medium Access Control
PWM	Pulse Width Modulation
RAM	Random Access Memory
RTS	Ready-To-Send
SI	Serial Interface
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver-Transmitter
WS	Word-Select

Related Documents

JN-UG-3066	Integrated Peripherals API User Guide
JN-RM-2003	Board API Reference Manual
JN-RM-2025	Application Queue API Reference Manual
JN-DS-JN5121	JN5121 Data Sheet
JN-DS-JN5139	JN5139 Data Sheet
JN-DS-JN5148	JN5148 Data Sheet

Feedback Address

If you wish to comment on this manual, or any other Jennic user documentation, please provide your feedback by writing to us (quoting the manual reference number and version) at the following postal address or e-mail address:

Applications
Jennic Ltd.
Furnival Street
Sheffield S1 4QT
United Kingdom

doc@jennic.com

1. Overview

This manual provides detailed descriptions of the C functions and related resources of the Integrated Peripherals Application Programming Interface (API) for the Jennic wireless microcontrollers. This API (sometimes referred to as the AHI) allows an application running on the wireless microcontroller to control the on-chip peripherals, which include:

- System Controller
- Analogue Peripherals:
  - Analogue-to-Digital Converter (ADC)
  - Digital-to-Analogue Converters (DACs)
  - Comparators
- Digital Input/Output (DIO)
- Universal Asynchronous Receiver-Transmitters (UARTs)
- Timers
- Wake Timers
- Tick Timer
- Watchdog Timer [JN5148 only]
- Pulse Counters [JN5148 only]
- Serial Interface (2-wire):
  - SI Master
  - SI Slave [JN5148 only]
- Serial Peripheral Interface (SPI master)
- Intelligent Peripheral (IP) Interface (SPI slave)
- Digital Audio Interface (DAI) [JN5148 only]
- Sample FIFO Interface [JN5148 only]
- External Flash memory

Note: You should refer to this manual in conjunction with the Jennic *Integrated Peripherals API User Guide (JN-UG-3066)*, which provides guidance on using the API functions to control the JN5148/JN5139 on-chip peripherals.

The functions of this API are defined in the header file **AppHardwareApi.h** and the software invoked by this API is located in the on-chip ROM.

Caution: *The Integrated Peripherals API functions described in this manual are not re-entrant functions. A function must be allowed to complete before that function is called again, otherwise unexpected results may occur.*

Note that the Integrated Peripherals API does NOT include functions to control:

- IEEE 802.15.4 MAC hardware built into the wireless microcontroller - this hardware is controlled by the wireless network protocol stack software (which may be an IEEE 802.15.4, ZigBee, Jenie/JenNet or 6LoWPAN/JenNet stack), and APIs for this purpose are provided with the appropriate stack software product.
- Resources of the Jennic evaluation kit boards, such as sensors and display panels (although the buttons and LEDs on the Jennic evaluation kit boards are connected to the DIO pins of the wireless microcontroller) - a special function library, called the Board API, is provided by Jennic for this purpose and is described in the *Board API Reference Manual (JN-RM-2003)*.

2. General Functions

This chapter describes various functions of the Integrated Peripherals API that are not associated with any of the main peripheral blocks on a Jennic wireless microcontroller. Note that many of the functions described in this chapter can be used only on the JN5148 device.

The functions in this chapter include:

- API initialisation function
- Functions concerned with radio transmissions (including setting the transmission power and data-rate)
- Functions to control the random number generator (JN5148 only)
- Stack overflow detection function

Note that the random number generator can produce interrupts which are treated as System Controller interrupts. For more information on interrupt handling, refer to Appendix A.

The functions are listed below, along with their page references:

Function	Page
u32AHI_Init	18
bAHI_PhyRadioSetPower (JN5139/JN5148 Only)	19
vAppApiSetBoostMode (JN5139 Only)	20
vAHI_HighPowerModuleEnable	21
vAHI_ETSIHighPowerModuleEnable (JN5148 Only)	22
vAHI_AntennaDiversityOutputEnable	23
vAHI_BbcSetHigherDataRate (JN5148 Only)	24
vAHI_BbcSetInterFrameGap (JN5148 Only)	25
vAHI_StartRandomNumberGenerator (JN5148 Only)	26
vAHI_StopRandomNumberGenerator (JN5148 Only)	27
u16AHI_ReadRandomNumber (JN5148 Only)	28
bAHI_RndNumPoll (JN5148 Only)	29
vAHI_SetStackOverflow (JN5148 Only)	30

Note: For guidance on using these functions in JN5148/JN5139 application code, refer to Chapter 2 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

u32AHI_Init

```
uint32 u32AHI_Init(void);
```

Description

This function initialises the Integrated Peripherals API. It should be called after every reset and wake-up, and before any other Integrated Peripherals API functions are called.

Caution: *If you are using JenOS (Jennic Operating System), you must not call this function explicitly in your code, as the function is called internally by JenOS. This applies principally to users who are developing ZigBee PRO applications.*

Note: This function must be called before initialising the Application Queue API (if used). For more information on the latter, refer to the *Application Queue API Reference Manual (JN-RM-2025)*.

Parameters

None

Returns

0 if initialisation failed, otherwise a 32-bit version number for the API (most significant 16 bits are main revision, least significant 16 bits are minor revision).

bAHI_PhyRadioSetPower (JN5139/JN5148 Only)

bool_t bAHI_PhyRadioSetPower(uint8 *u8PowerLevel*);

Description

This function sets the transmit power level of the JN5148/JN5139 device's radio transceiver. The levels that can be set depend on the type of module (JN5139 standard, JN5139 high-power, JN5148 standard, JN5148 high-power), as indicated in the table below:

***u8PowerLevel* Setting**	**Power Level (dBm)**			
	JN5139 Modules		**JN5148 Modules**	
	Standard	**High-Power**	**Standard**	**High-Power**
0	-30	-7	-32	-16.5
1	-24	-1	-20.5	-5
2	-18	+5	-9	+6.5
3	-12	+11	+2.5	+18
4	-6	+15	-	-
5	+1.5	+17.5	-	-

Note that the above power levels are nominal values. The actual power levels obtained vary with temperature and supply voltage. The quoted values are typical for an operating temperature of 25^{o}C and a supply voltage of 3.0 V.

Before this function is called, **vAHI_ProtocolPower()** must have been called.

Before using a high-power module, its radio transceiver must be enabled via the function **vAHI_HighPowerModuleEnable()**.

Parameters

u8PowerLevel	Integer value in the range 0-5 representing the desired radio power level (the default value is 5 for JN5139 modules and 3 for JN5148 modules). The corresponding power levels (in dBm) depend on the type of JN5148/JN5139 module and are detailed in the above table. Note that values 4 and 5 are not valid for JN5148 modules

Returns

One of:

TRUE if specified power setting is valid (in the range 0-5)

FALSE if specified power setting is invalid (not in the range 0-5)

vAppApiSetBoostMode (JN5139 Only)

```
void vAppApiSetBoostMode(bool_t bOnNotOff);
```

Description

This function enables or disables boost mode on a JN5139 device. Boost mode increases the radio transmission power by 1.5 dBm (beware that this results in increased current consumption). This feature can only be used with standard JN5139 modules (and not high-power modules), thus increasing the maximum possible transmit power to +3 dBm.

If required, this function must be the very first call in your code. A new setting only takes effect when the device is initialised, so this function must be called before intialising the stack and before calling **u32AppQApiInit()** (if the Application Queue API is used). The setting is maintained throughout sleep if memory is held, but is lost if memory is not held during sleep.

Parameters

bOnNotOff	On/off setting for boost mode: TRUE - enable boost mode FALSE - disable boost mode (default setting)

Returns

None

vAHI_HighPowerModuleEnable

```
void vAHI_HighPowerModuleEnable(bool_t bRFTXEn,
                                bool_t bRFRXEn);
```

Description

This function allows the transmitter and receiver sections of a Jennic high-power module to be enabled or disabled. The transmitter and receiver sections must both be enabled or disabled at the same time (enabling only one of them is not supported). The function must be called before using the radio transceiver on a high-power module.

The function sets the CCA (Clear Channel Assessment) threshold to suit the gain of the attached Jennic high-power module.

Caution: *A Jennic high-power module cannot be used in channel 26 of the 2.4-GHz band.*

Note that this function cannot be used with a high-power module from a manufacturer other than Jennic.

The European Telecommunications Standards Institute (ETSI) dictates an operating power limit for Europe of +10 dBm EIRP. If you wish to operate a JN5148 high-power module close to this power limit, you should subsequently call the function **vAHI_ETSIHighPowerModuleEnable()**.

Parameters

bRFTXEn	Enable/disable setting for high-power module transmitter (must be same setting as for *bRFRXEn*): TRUE - enable transmitter FALSE - disable transmitter
bRFRXEn	Enable/disable setting for high-power module receiver (must be same setting as for *bRFTXEn*): TRUE - enable receiver FALSE - disable receiver

Returns

None

vAHI_ETSIHighPowerModuleEnable (JN5148 Only)

void vAHI_ETSIHighPowerModuleEnable(bool_t *bOnNotOff***);**

Description

This function sets the power output of a JN5148 high-power module just within the limit of +10 dBm EIRP dictated by the European Telecommunications Standards Institute (ETSI). The function sets the power output of the module to +8 dBm, which is suitable for use with an antenna with a gain of up to +2 dBi.

Before calling this function, the transmitter of the high-power module must be enabled using the function **vAHI_HighPowerModuleEnable()**.

Note that this function cannot be used with a high-power module from a manufacturer other than Jennic.

Parameters

bOnNotOff	Enable/disable ETSI power limit on high-power module: TRUE - enable limit FALSE - disable limit (returns to normal high-power module setting)

Returns

None

vAHI_AntennaDiversityOutputEnable

```
void vAHI_AntennaDiversityOutputEnable(
                              bool_t bOddRetryOutEn,
                              bool_t bEvenRetryOutEn);
```

Description

This function can be used to individually enable or disable the use of DIO12 (pin 53) and DIO13 (pin 54) to control up to two antennae when packets are re-transmitted following an initial transmission failure.

The JN5148 has two antenna diversity outputs, on DIO12 and DIO13, but the JN5121 and JN5139 devices only have one antenna diversity output, on DIO12. Therefore, the parameter *bEvenRetryOutEn* (for DIO13) is only applicable to JN5148, and should be set to FALSE for JN5121 and JN5139.

Refer to your device datasheet for more information on the antenna diversity output.

Parameters

bOddRetryOutEn	Enable/disable setting for DIO12: TRUE - enable output on pin FALSE - disable output on pin
bEvenRetryOutEn	Enable/disable setting for DIO13 (JN5148 only): TRUE - enable output on pin FALSE - disable output on pin

Returns

None

vAHI_BbcSetHigherDataRate (JN5148 Only)

void vAHI_BbcSetHigherDataRate(uint8 *u8DataRate***);**

Description

This function sets the data-rate for over-air radio transmissions from the JN5148 device. Before this function is called, **vAHI_ProtocolPower()** must have been called.

The standard data-rate is 250 Kbps but one of two alternative rates can be set using this function: 500 Kbps and 666 Kbps. Note that these alternatives are not standard IEEE 802.15.4 modes and performance in these modes is degraded by at least 3 dB. There will be a residual error-rate caused by any frequency offset when operating at 666 Kbps.

Provision of the alternative data-rates allows on-demand, burst transmissions between nodes.

Note that the data-rate set by this function does not only apply to data transmission, but also to data reception - the device will only be able to receive data sent at the rate specified through this function. Therefore, this data-rate must be also be taken into account by the sending node.

Parameters

u8DataRate	Data rate to set: E_AHI_BBC_CTRL_DATA_RATE_250_KBPS (250 Kbps) E_AHI_BBC_CTRL_DATA_RATE_500_KBPS (500 Kbps) E_AHI_BBC_CTRL_DATA_RATE_666_KBPS (666 Kbps)

Returns

None

vAHI_BbcSetInterFrameGap (JN5148 Only)

void vAHI_BbcSetInterFrameGap(uint8 *u8Lifs***);**

Description

This function sets the long inter-frame gap for over-air radio transmissions of IEEE 802.15.4 frames from the JN5148 device. Before this function is called, **vAHI_ProtocolPower()** must have been called and the radio section of the JN5148 chip must have been initialised (done when the protocol stack is started).

The long inter-frame gap must be a multiple of 4 µs and this function multiplies the specified value (*u8Lifs*) by 4 ìs to obtain the long inter-frame gap to be set.

The standard long inter-frame gap (as specified by IEEE 802.15.4) is 640 µs. Reducing it may result in an increase in the throughput of frames. The recommended minimum value is 192 µs. The function imposes a lower limit of 184 µs on the long inter-frame gap, so it is not possible to achieve a value below this limit, irrespective of the setting in this function.

The function can be used to configure two nodes to exchange messages by means of non-standard transmissions. To maintain compliance with the IEEE 802.15.4 standard, this function should not be called.

Parameters

u8Lifs	Long inter-frame gap, in units of 4 microseconds (e.g. for a gap of 192 µs, set this parameter to 48). Specifying a value of less than 46 results in a setting of 46, corresponding to 184 µs

Returns

None

vAHI_StartRandomNumberGenerator (JN5148 Only)

```
void vAHI_StartRandomNumberGenerator(
                                    bool_t const bMode,
                                    bool_t const bIntEn);
```

Description

This function starts the random number generator on the JN5148 device, which produces 16-bit random values. The generator can be started in one of two modes:

- **Single-shot mode:** Stop generator after one random number
- **Continuous mode:** Run generator continuously - this will generate a random number every 256 µs

A randomly generated value can subsequently be read using the function **u16AHI_ReadRandomNumber()**. The availability of a new random number, and therefore the need to call the 'read' function, can be determined using either interrupts or polling:

- When random number generator interrupts are enabled, an interrupt will occur each time a new random value is generated. These interrupts are handled by the callback function registered with **vAHI_SysCtrlRegisterCallback()** - also refer to Appendix A.
- Alternatively, when random number generator interrupts are disabled, the function **bAHI_RndNumPoll()** can be used to poll for the availability of a new random value.

When running continuously, the random number generator can be stopped using the function **vAHI_StopRandomNumberGenerator()**.

Note that the random number generator uses the 32-kHz clock domain (see Chapter 3) and will not operate properly if a high-precision external 32-kHz clock source is used. Therefore, if generating random numbers in your application, you are advised to use the internal RC oscillator or a low-precision external clock source.

Parameters

bMode	Generator mode: E_AHI_RND_SINGLE_SHOT (single-shot mode) E_AHI_RND_CONTINUOUS (continuous mode)

bIntEn	Enable/disable interrupts setting:
	E_AHI_INTS_ENABLED(enable)
	E_AHI_INTS_DISABLED(disable)

Returns

None

vAHI_StopRandomNumberGenerator (JN5148 Only)

```
void vAHI_StopRandomNumberGenerator(void);
```

Description

This function stops the random number generator on the JN5148 device, if it has been started in continuous mode using **vAHI_StartRandomNumberGenerator()**.

Parameters

None

Returns

None

u16AHI_ReadRandomNumber (JN5148 Only)

```
uint16 u16AHI_ReadRandomNumber(void);
```

Description

This function obtains the last 16-bit random value produced by the random number generator on the JN5148 device. The function can only be called once the random number generator has generated a new random number.

The availability of a new random number, and therefore the need to call **u16AHI_ReadRandomNumber()**, is determined using either interrupts or polling:

- When random number generator interrupts are enabled, an interrupt will occur each time a new random value is generated.
- Alternatively, when random number generator interrupts are disabled, the function **bAHI_RndNumPoll()** can be used to poll for the availability of a new random value.

Interrupts are enabled or disabled when the random number generator is started using **vAHI_StartRandomNumberGenerator()**.

Parameters

None

Returns

16-bit random integer

bAHI_RndNumPoll (JN5148 Only)

```
bool_t bAHI_RndNumPoll(void);
```

Description

This function can be used to poll the random number generator on the JN5148 device - that is, to determine whether the generator has produced a new random value.

Note that this function does not obtain the random value, if one is available - the function **u16AHI_ReadRandomNumber()** must be called to read the value.

Parameters

None

Returns

Availability of new random value, one of:

TRUE - random value available
FALSE - no random value available

vAHI_SetStackOverflow (JN5148 Only)

```
void vAHI_SetStackOverflow(bool_t bStkOvfEn,
                           uint32 u32Addr);
```

Description

This function allows stack overflow detection to be enabled/disabled on the JN5148 device and a threshold to be set for the generation of a stack overflow exception.

The JN5148 processor has a stack for temporary storage of data during code execution, such as local variables and return addresses from functions. The stack begins at the highest location in RAM (0x04020000) and grows downwards through RAM, as required. Thus, the stack size is dynamic, typically growing when a function is called and shrinking when returning from a function. It is difficult to determine by code inspection exactly how large the stack may grow. The lowest memory location currently used by the stack is stored in the stack pointer.

Applications occupy the bottom region of RAM and the memory space required by the applications is fixed at build time. Above the applications is the heap, which is used to store static data. The heap grows upwards through RAM as data is added. Since the actual space needed by the processor stack is not known at build time, it is possible for the processor stack to grow downwards into the heap space while the application is running. This condition is called a stack overflow and results in the processor stack corrupting the heap (and potentially the application).

This function allows a threshold RAM address to be set, such that a stack overflow exception is generated if and when the stack pointer falls below this threshold address. The threshold address is specified as a 17-bit offset from the base of RAM (from 0x04000000). It can take values in the range 0x00000 to 0x1FFFC (the stack pointer is word-aligned, so the bottom 2 bits of the address are always 0). The value 0x1F800 is a good starting point.

Note 1: If a stack overflow is detected, the detection mechanism is automatically disabled and this function must be called to re-enable it.

Note 2: An exception handler should be developed and configured before enabling stack overflow detection.

Parameters

bStkOvfEn	Enable/disable stack overflow detection: TRUE - enable detection FALSE - disable detection (default)
u32Addr	17-bit stack overflow threshold, in range 0x00000 to 0x1FFFC

Returns

None

3. System Controller

This chapter describes the functions that interface to the system controller on the Jennic wireless microcontrollers. The system controller is largely concerned with controlling the power domains for the CPU and on-chip RAM, and has a key role in implementing low-power sleep modes.

The functions detailed in this chapter cover the following areas:

- Power management
- Clock management
- Voltage brownout
- Chip reset

System Controller Blocks

There are two main blocks in the system controller - a 16-MHz and a 32-kHz domain:

- **16-MHz Domain:** This domain is used to run the CPU and most peripherals (e.g. security engine, ADC, DACs, UARTs, timers and SPI master) when the chip is fully operational. The 16-MHz clock is sourced as follows, dependent on the chip type:
  - JN5148: 32-MHz crystal oscillator or 24-MHz RC oscillator (see footnote*)
  - JN5139/JN5121: 16-MHz crystal oscillator

  The crystal oscillators are driven from external crystals of the relevant frequencies connected to pins 8 and 9 for JN5148, and pins 11 and 12 for JN5139/JN5121.

  For JN5148, the CPU clock is not fixed at 16 MHz, but can be configured to run at 4, 8, 16 or 32 MHz (if sourced from the 32-MHz crystal oscillator) or at 3, 6, 12 or 24 MHz (if sourced from the 24-MHz RC oscillator).
- **32-kHz Domain:** This domain is designed for very low-power sleep states. While in such sleep states, the CPU does not run and relies on an interrupt to wake it up. The interrupt can be generated externally or from within the 32-kHz domain. Later chapters include ways in which such interrupts can be generated within the chip, using the wake timers to generate events within the 32-kHz domain or from external stimuli via the DIO pins. The 32-kHz clock can be sourced as follows, dependent on the chip type:
  - JN5148: Internal RC oscillator, external crystal or external clock module
  - JN5139: Internal RC oscillator or external clock module
  - JN5121: Internal RC oscillator

  For JN5148, the crystal oscillator is driven from an external 32-kHz crystal connected to DIO9 and DIO10 (pins 50 and 51). For JN5148 and JN5139, the external clock is connected to DIO9 (pin 50).

The 32-kHz domain is still active when the chip is operating normally (it is used for the random number generator on the JN5148 device - see Chapter 2) and can be calibrated against the 16-MHz clock to improve timing accuracy.

* Note that the 24-MHz RC oscillator will produce a 12-MHz clock (and not a 16-MHz clock).

Power Domains and Lower-Power Modes

The Jennic wireless microcontrollers have a number of independent power domains. For a full description of these domains, refer to the Jennic data sheet for your wireless microcontroller.

The CPU and on-chip RAM are powered by separate voltage regulators. This allows flexibility in implementing different low-power sleep modes. The application is loaded from (external) Flash memory into on-chip RAM when the wireless microcontroller powers up. During sleep mode (while the CPU is powered down), the following options exist for the application:

- In order to minimise current consumption during sleep, RAM power can also be removed. In this case, the application will be lost and it will be necessary to re-load the application from Flash memory on waking.
- RAM can remain powered (by its own regulator), allowing the application to be retained in RAM during low-power sleep periods. This is useful for short sleep periods, when the time taken on waking to re-load the application from Flash memory to RAM is significant compared with the sleep duration.

This leads to the following possible sleep modes, in which the CPU is powered down:

- **Sleep with memory held and 32-kHz oscillator on:** RAM remains powered and the 32-kHz oscillator remains on.
- **Sleep without memory held and 32-kHz oscillator on:** RAM is powered down but the 32-kHz oscillator remains on.
- **Sleep with memory held and 32-kHz oscillator off (not JN5121):** RAM remains powered but the 32-kHz oscillator switched off.
- **Sleep without memory held and 32-kHz oscillator off (not JN5121):** RAM is powered down and the 32-kHz oscillator switched off.
- **Deep sleep:** Both the 16-MHz and 32-kHz clock domains are switched off and the device can only be woken by the device’s reset line being pulled low (all chips) or an external event which triggers a change on a DIO pin (JN5139 and JN5148 only).

Doze mode is another alternative in which the CPU, radio transceiver and digital peripherals remain powered but the clock to the CPU is stopped (although the rest of the 16-MHz domain remains powered, as do RAM and other power domains). This mode uses more power than sleep mode but requires less time to restart. The CPU is brought out of doze mode by an interrupt (note that a tick-timer interrupt cannot be used to bring the CPU out of doze mode on the JN5121 and JN5139 devices).

The system controller functions are listed below, along with their page references:

Function	**Page**
u8AHI_PowerStatus	34
vAHI_MemoryHold	35
vAHI_CpuDoze	36
vAHI_PowerDown (JN5121 Only)	37
vAHI_Sleep (JN5139/JN5148 Only)	38
vAHI_ProtocolPower	40
vAHI_ExternalClockEnable (JN5139 Only)	41
bAHI_Set32KhzClockMode (JN5148 Only)	42
vAHI_SelectClockSource (JN5148 Only)	43
bAHI_GetClkSource (JN5148 Only)	44
bAHI_SetClockRate (JN5148 Only)	45
u8AHI_GetSystemClkRate (JN5148 Only)	46
vAHI_EnableFastStartUp (JN5148 Only)	47
vAHI_PowerXTAL (JN5148 Only)	48
vAHI_BrownOutConfigure (JN5148 Only)	49
bAHI_BrownOutStatus (JN5148 Only)	51
bAHI_BrownOutEventResetStatus (JN5148 Only)	52
u32AHI_BrownOutPoll (JN5148 Only)	53
vAHI_SwReset	54
vAHI_DriveResetOut	55
vAHI_ClearSystemEventStatus	56
vAHI_SysCtrlRegisterCallback	57

Note: For guidance on using these functions in JN5148/ JN5139 application code, refer to Chapter 3 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

u8AHI_PowerStatus

```
uint8 u8AHI_PowerStatus(void);
```

Description

This function returns power domain status information for the wireless microcontroller - in particular, whether:

- The device has completed a sleep-wake cycle
- RAM contents were retained during sleep
- The analogue power domain is switched on
- The protocol power domain is switched on (JN5121 only)
- The protocol logic is operational - clock is enabled (JN5148/JN5139 only)

Parameters

None

Returns

Returns the power domain status information in bits 0-3 of the 8-bit return value:

Bit	Reads a ‘1’ if...
0	Device has completed a sleep-wake cycle
1	RAM contents were retained during sleep
2	Analogue power domain is switched on
3	• Protocol power domain is switched on (JN5121 only) • Protocol logic is operational (JN5148/JN5139 only)
4-7	Unused

vAHI_MemoryHold

void vAHI_MemoryHold(bool_t *bHoldDuringSleep***);**

Description

This function can be used to control whether on-chip RAM will remain powered (and therefore retain its contents) during any subsequent sleep periods. In particular, the function is used for sleep periods initiated using the function **vAHI_PowerDown()** on the JN5121 device.

Caution: *The* ***vAHI_MemoryHold()*** *function is available but not recommended for use on the JN5148 and JN5139 devices, as these devices are put to sleep using the function* ***vAHI_Sleep()*** *which has its own provision for controlling the power to on-chip RAM during sleep.*

Parameters

bHoldDuringSleep	Power status of RAM for sleep periods: TRUE to power memory during sleep FALSE to remove power from memory during sleep

Returns

None

vAHI_CpuDoze

```
void vAHI_CpuDoze(void);
```

Description

This function puts the device into doze mode by stopping the clock to the CPU (other on-chip components are not affected by this functon and so will continue to operate normally, e.g. on-chip RAM will remain powered and so retain its contents). The CPU will cease operating until an interrupt occurs to re-start normal operation. Disabling the CPU clock in this way reduces the power consumption of the device during inactive periods.

Note: Tick Timer interrupts can be used to wake the CPU from doze mode on the JN5148 device, but not on the JN5139 and JN5121 devices.

The function returns when the CPU re-starts.

Parameters

None

Returns

None

vAHI_PowerDown (JN5121 Only)

void vAHI_PowerDown(bool_t *bDeepNotNormalSleep***);**

Description

This function puts the device into sleep mode, being one of:

- **Deep sleep mode:** In this mode, all components of the chip are powered down (including the 32-kHz and 16-MHz oscillators). The device can be woken by the device's reset line being pulled low.
- **'Normal' sleep mode:** In this mode, RAM is powered down but the 32-kHz oscillator remains on. The device can be woken by an interrupt (DIO or wake timer) or a reset. If you wish to preserve RAM contents while sleeping then before calling this function you must call the function **vAHI_MemoryHold()** to ensure that RAM remains powered.

A `while(1);` should be placed immediately after this function is executed, as the function may return before the CPU has powered down. In addition, all pending interrupts must be cleared before this function is called. When the device restarts, it will perform a cold start, unless the device is exiting 'normal' sleep with memory held (**vAHI_MemoryHold()** was called), in which case it will perform a warm start.

Note: Registered callback functions are only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, any callback functions must be re-registered before calling **u32AHI_Init()** on waking. Alternatively, a DIO wake source can be resolved using **u32AHI_DioWakeStatus()**.

Parameters

bDeepNotNormalSleep	Required sleep mode, one of: TRUE for deep sleep mode FALSE for normal sleep mode

Returns

None

vAHI_Sleep (JN5139/JN5148 Only)

void vAHI_Sleep(teAHI_SleepMode *sSleepMode***);**

Description

This function puts the JN5148/JN5139 device into sleep mode, being one of four 'normal' sleep modes or deep sleep mode. The normal sleep modes are distinguished by whether on-chip RAM remains powered and whether the 32-kHz oscillator is left running during sleep (see parameter description below).

Note 1: If an external source is used for the 32-kHz oscillator on the JN5148 device (see page 31), it is not recommended that the oscillator is stopped on entering sleep mode.

Note 2: Registered callback functions are only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, any callback functions must be re-registered before calling **u32AHI_Init()** on waking. Alternatively, a DIO wake source can be resolved using **u32AHI_DioWakeStatus()**.

Note 3: If a JN5148 high-power module is being used, this function will power down lines to the high-power module that draw significant current.

- In a normal sleep mode, the device can be woken by a reset or one of the following interrupts:
  - DIO interrupt
  - Wake timer interrupt (needs 32-kHz oscillator to be left running during sleep)
  - Comparator interrupt
  - Pulse counter interrupt (JN5148 only - see introduction to Chapter 11)

  External Flash memory is not powered down during normal sleep mode. On the JN5148 and JN5139 devices, if required, you can power down the Flash memory device using the function **vAHI_FlashPowerDown()**, which must be called before **vAHI_Sleep()**, provided you are using a compatible Flash memory device - refer to the description of **vAHI_FlashPowerDown()** on page 275.
- In deep sleep mode, all components of the chip are powered down, as well as external Flash memory, and the device can only be woken by the device's reset line being pulled low or an external event which triggers a change on a DIO pin (the relevant DIO must be configured as an input and DIO interrupts must be enabled).

When the device restarts, it will begin processing at the cold start or warm start entry point, depending on the sleep mode from which the device is waking (see below).
This function does not return.

Parameters

sSleepMode	Required sleep mode, one of: E_AHI_SLEEP_OSCON_RAMON 32-kHz oscillator on and RAM on (warm restart) E_AHI_SLEEP_OSCON_RAMOFF 32-kHz oscillator on and RAM off (cold restart) E_AHI_SLEEP_OSCOFF_RAMON 32-kHz oscillator off and RAM on (warm restart) E_AHI_SLEEP_OSCOFF_RAMOFF 32-kHz oscillator off and RAM off (cold restart) E_AHI_SLEEP_DEEP Deep sleep (all components off - cold restart)

Returns

None

vAHI_ProtocolPower

void vAHI_ProtocolPower(bool_t *bOnNotOff***);**

Description

This function is used as follows on the different wireless microcontrollers:

- **JN5121:** To enable or disable the voltage regulator that supplies the Protocol Power domain
- **JN5139/JN5148:** To enable or disable the clock for the Digital Logic domain - the clock is simply disabled (gated) while the domain remains powered

If you intend to switch the regulator/clock off and then back on again, without performing a reset or going through a sleep cycle, you must first save the current IEEE 802.15.4 MAC settings before switching off the regulator/clock. Upon switching the regulator/clock on again, the MAC settings must be restored from the saved settings. You can save and restore the MAC settings using functions of Jennic's 802.15.4 Stack API:

- To save the MAC settings, use the function **vAppApiSaveMacSettings()**.
- Switching the regulator/clock back on can then be achieved by restoring the MAC settings using the function **vAppApiRestoreMacSettings()** (this function automatically calls **vAHI_ProtocolPower()** to switch on the regulator/clock)

The MAC settings save and restore functions are described in the *802.15.4 Stack API Reference Manual (JN-RM-2002)*.

While the regulator/clock is off, you must not make any calls into the stack, as this may result in the stack attempting to access the associated hardware (which is disabled) and therefore cause an exception.

Caution: *Do not call* ***vAH_ProtocolPower(FALSE)*** *while the 802.15.4 MAC layer is active, otherwise the device may freeze.*

Parameters

bOnNotOff	Setting for regulator (JN5121) or clock (JN5139/JN5148): TRUE to switch the regulator/clock ON FALSE to switch the regulator/clock OFF

Returns

None

vAHI_ExternalClockEnable (JN5139 Only)

void vAHI_ExternalClockEnable(bool_t *bExClockEn***);**

Description

This function can be used to enable the use of an external source for the 32-kHz clock on a JN5139 device (the function is used to move from the internal source to an external source). The function should be called only following a device reset and not following a wake-up from sleep (since this clock selection is maintained during sleep).

The external clock must be supplied on DIO9 (pin 50), with the other end tied to ground. Note that there is no need to explicitly configure DIO9 as an input, as this is done automatically by the function. However, you are advised to first disable the pull-up on this DIO using the function **vAHI_DioSetPullup()**.

If this function is not called, the internal 32-kHz RC oscillator is used by default.

Once this function has been called to enable an external clock input, you are not advised to subsequently change back to the internal oscillator.

Note that the equivalent function for the JN5148 device is **bAHI_Set32KhzClockMode()** and there is no option to use an external 32-kHz clock on the JN5121 device.

Parameters

bExClockEn	Enable/disable setting for external 32-kHz clock: TRUE - enable external clock input FALSE - disable external clock input

Returns

None

bAHI_Set32KhzClockMode (JN5148 Only)

bool_t bAHI_Set32KhzClockMode(uint8 const *u8Mode***);**

Description

This function selects an external source for the 32-kHz clock for the JN5148 device (the function is used to move from the internal source to an external source). The selected clock can be either of the following options:

- **External module (RC circuit):** This clock must be supplied on DIO9 (pin 50)
- **External crystal:** This circuit must be attached on DIO9 (pin 50) and DIO10 (pin 51)

If this function is not called, the internal 32-kHz RC oscillator is used by default. Note that once an external 32-kHz clock source has been selected using this function, it is not possible to switch back to the internal RC oscillator.

If required, this function should be called near the start of the application. In particular, if selecting the external crystal, the function must be called before Timers 0 and 1, and any wake timers are used by the application, since these timers are used by the function when switching the clock source to the external crystal.

Caution: *When switching to an external crystal, this function automatically takes control of the DIOs (11, 12 and 13) associated with Timer 1 unless the application first makes the call* ***vAHI_TimerDIOControl(****E_AHI_TIMER1, FALSE***)***. Also, the function does not disable Timer 1 following the switch - Timer 1 should then be disabled by the application through the call* ***vAHI_TimerDisable(****E_AHI_TIMER1***)***.*

Note that there is no need to explicitly configure DIO9 or DIO10 as an input, as this is done automatically by the function.

When selecting an external module, you must disable the pull-up on DIO9 using the function **vAHI_DioSetPullup()**. However, when selecting the external crystal, the pull-ups on DIO9 and DIO10 are disabled automatically.

Note that the equivalent function for the JN5139 device is **vAHI_ExternalClockEnable()** but there is no option to use an external 32-kHz clock on the JN5121 device.

Parameters

u8Mode	External 32-kHz clock source: E_AHI_EXTERNAL_RC (external module) E_AHI_XTAL (external crystal)

Returns

Validity of specified clock source, one of:
TRUE - valid clock source specified
FALSE - invalid clock source specified

vAHI_SelectClockSource (JN5148 Only)

```
void vAHI_SelectClockSource(bool_t bClkSource,
                            bool_t bPowerDown);
```

Description

This function selects the clock source for the system clock on the JN5148 device. The clock options are:

- 32-MHz crystal oscillator (XTAL), derived from external crystal on pins 8 and 9
- 24-MHz RC oscillator

The clock source is divided by two to produce the system clock. Thus, the crystal oscillator will produce a 16-MHz system clock and the RC oscillator will produce a 12-MHz (±30%, unless calibrated) system clock (see Caution below).

***Caution:** You will not be able to run the full system while using the 24-MHz clock source. It is possible to execute code while using this clock, but it is not possible to transmit or receive. Further, calculated baud rates and timing intervals for the UARTs and timers should be based on 12 MHz. You are also not advised to change from the crystal oscillator to the RC oscillator.*

When the RC oscillator is selected, the function allows the crystal oscillator to be powered down, in order to save power.

If the crystal oscillator is selected using this function but the oscillator is not already running when the function is called (see **vAHI_EnableFastStartUp()**), at least 1 ms will be required for the oscillator to become stable once it has powered up. The function will not return until the oscillator has stabilised.

The function is mainly useful in conjunction with **vAHI_EnableFastStartUp()** to perform a manual switch from the RC oscillator to the crystal oscillator after sleeping.

Parameters

bClkSource	System clock source: TRUE - RC oscillator FALSE - crystal oscillator
bPowerDown	Power down crystal oscillator: TRUE - power down when not needed FALSE - leave powered up (when not in sleep mode)

Returns

None

bAHI_GetClkSource (JN5148 Only)

```
bool_t bAHI_GetClkSource(void);
```

Description

This function obtains the identity of the clock source for the system clock. The clock options are:

- 32-MHz crystal oscillator (XTAL), derived from external crystal on pins 8 and 9
- 24-MHz RC oscillator

Parameters

None

Returns

Clock source, one of:
TRUE - 24-MHz RC oscillator
FALSE - 32-MHz crystal oscillator

bAHI_SetClockRate (JN5148 Only)

```
bool_t bAHI_SetClockRate(uint8 u8Speed);
```

Description

This function is used to select a CPU clock rate on the JN5148 device by setting the divisor used to derive the CPU clock from the system clock.

The system clock source is selected as either the 32-MHz external crystal oscillator or the 24-MHz internal RC oscillator using the function **vAHI_SelectClockSource()**.

Parameters

u8Speed Divisor for desired CPU clock frequency:

u8Speed	Clock Divisor	Resulting Frequency	
		From 32 MHz	From 24 MHz
000	8	4 MHz	3 MHz
001	4	8 MHz	6 MHz
010	2	16 MHz	12 MHz
011	1	32 MHz	24 MHz
100-111	Invalid		

Note: When the 24-MHz RC oscillator is used as the source, the resulting CPU clock frequency is dictated by the actual RC oscillator frequency, which can be 24 MHz ±30%.

Returns

TRUE if successful, FALSE if invalid clock frequency specified (100-111)

u8AHI_GetSystemClkRate (JN5148 Only)

```
uint8 u8AHI_GetSystemClkRate(void);
```

Description

This function obtains the divisor used to divide down the system clock source to produce the CPU clock.

The system clock source is selected as either the 32-MHz external crystal oscillator or the 24-MHz internal RC oscillator using the function **vAHI_SelectClockSource()**. The clock source can be obtained using the function **bAHI_GetClkSource().**

The CPU clock frequency can be calculated by dividing the source clock frequency by the returned divisor. The results are summarised in the table below.

Returned Value	Clock Divisor	Resulting Frequency	
		From 32 MHz	From 24 MHz
000	8	4 MHz	3 MHz
001	4	8 MHz	6 MHz
010	2	16 MHz	12 MHz
011	1	32 MHz	24 MHz
100-111	Invalid		

Note: When the 24-MHz RC oscillator is used as the source, the resulting CPU clock frequency is dictated by the actual RC oscillator frequency, which can be 24 MHz ±30%.

The divisor for the CPU clock is configured using the function **bAHI_SetClockRate()**.

Parameters

None

Returns

Clock divisor:
- 000: Divisor of 8
- 001: Divisor of 4
- 010: Divisor of 2
- 011: Divisor of 1 (source frequency untouched)

vAHI_EnableFastStartUp (JN5148 Only)

void vAHI_EnableFastStartUp(bool_t *bMode*,
bool_t *bPowerDown***);**

Description

This function can be used to enable fast start-up of the JN5148 device when waking from sleep. The function is relevant to a sleeping device for which the system clock is derived from the 32-MHz crystal oscillator (the default clock source).

The 32-MHz crystal oscillator is powered down during sleep and takes some time to become available again when the JN5148 device wakes. A more rapid start-up from sleep can be achieved by using the 24-MHz RC oscillator immediately on waking and then switching to the 32-MHz crystal oscillator when it becomes available. This allows initial processing at wake-up to proceed before the 32-MHz clock is ready.

The switch to the 32-MHz clock source can be either automatic or manual:

- **Automatic switch:** The crystal oscillator starts immediately on waking from sleep (irrespective of the setting of the *bPowerDown* parameter - see below), allowing it to warm up and stabilise while the boot code is running. The crystal oscillator is then automatically and seamlessly switched to when ready. To determine whether the switch has taken place, you can use the function **bAHI_GetClkSource()**.
- **Manual switch:** The switch to the crystal oscillator takes place at any time the application chooses, using the function **vAHI_SelectClockSource()**. If the crystal oscillator is not already running when this manual switch is initiated, the oscillator will be automatically started. Depending on the oscillator's progress towards stabilisation at the time of the switch request, there may be a delay of up to 1 ms before the crystal oscillator is stable and the switch takes place.

During the temporary period while the 24-MHz clock source is being used, you should not attempt to transmit or receive, and you can only use the JN5148 peripherals with special care - refer to the Caution on page 43.

You may wish to initially use the 24-MHz RC oscillator on waking and then manually switch to the 32-MHz crystal oscillator only when it becomes necessary to start transmitting/receiving. In this case, to conserve power, you can use the *bPowerDown* parameter to keep the crystal oscillator powered down until it is needed.

Parameters

bMode	Automatic/manual switch to 32-MHz clock: TRUE - automatic switch FALSE - manual switch
bPowerDown	Power down crystal oscillator: TRUE - power down when not needed FALSE - leave powered up (when not in sleep mode)

Returns

None

vAHI_PowerXTAL (JN5148 Only)

void vAHI_PowerXTAL(bool_t *bIsOn***);**

Description

This function can be used on the JN5148 device to enable or disable the power supply to a 32-MHz external crystal source for the 16-MHz system clock.

Typically, this function would be called on waking from sleep

The source of the 32-kHz clock must be selected using the function **bAHI_Set32KhzClockMode()**. If an external crystal oscillator is selected as the source, the latter function will automatically power up the oscillator. However, it is then necessary to wait a little time until the crystal oscillator is properly up and running. The function **vAHI_PowerXTAL()** can be called before the function **bAHI_Set32KhzClockMode()** in order to start the crystal oscillator in advance of its selection, so that the 32-kHz clock becomes available immediately after selection. This approach also allows other code to be executed while the oscillator is warming up between the calls to **vAHI_PowerXTAL()** and **bAHI_Set32KhzClockMode()**.

Parameters

bIsOn	Power setting for external crystal: TRUE to DISABLE power to crystal FALSE to ENABLE power to crystal

Returns

None

vAHI_BrownOutConfigure (JN5148 Only)

```
void vAHI_BrownOutConfigure(unit8 u8VboSelect,
                            bool_t bVboRestEn,
                            bool_t bVboEn,
                            bool_t bVboIntEnFalling,
                            bool_t bVboIntEnRising);
```

Description

This function configures and enables brownout detection on the JN5148 device.

Brownout is the point at which the chip supply voltage falls to (or below) a pre-defined level. The default brownout level is set to 2.3 V in the JN5148 device during manufacture. This function can be used to temporarily over-ride the default brownout voltage with one of four voltage levels (which include the default). There is a delay of up to 30 µs before the new setting will take effect.

The occurrence of the brownout condition is tracked by an internal ‘brownout bit’ in the device, which is set to:

- ‘1’ when the brownout state is entered - that is, when the supply voltage crosses the brownout voltage from above (decreasing supply voltage)
- ‘0’ when the brownout state is exited - that is, when the supply voltage crosses the brownout voltage from below (increasing supply voltage)

When brownout detection is enabled, the occurrence of a brownout event can be detected by the application in one of three ways:

- An automatic device reset (if configured using this function) - the function **bAHI_BrownOutEventResetStatus()** is used to check if a brownout caused a reset
- A brownout interrupt (if configured using this function) - see below
- Manual polling using the function **u32AHI_BrownOutPoll()**

Note: Following a device reset or sleep, ‘reset on brownout’ will be re-enabled and the default setting for the brownout voltage threshold will be re-instated.

Interrupts can be individually enabled that are generated when the chip goes into and out of brownout. Brownout interrupts are handled by the System Controller callback function, which is registered using the function **vAHI_SysCtrlRegisterCallback()**.

Parameters

u8VboSelect	Voltage threshold for brownout: 0: 2.0 V 1: 2.3 V 2: 2.7 V 3: 3.0 V

bVboRestEn	Enable/disable 'reset on brownout': TRUE to enable reset FALSE to disable reset
bVboEn	Enable/disable brownout detection: TRUE to enable detection FALSE to disable detection
bVboIntEnFalling	Enable/disable interrupt generated when the brownout bit falls, indicating that the device has come out of the brownout state: TRUE to enable interrupt FALSE to disable interrupt
bVboIntEnRising	Enable/disable interrupt generated when the brownout bit rises, indicating that the device has entered the brownout state: TRUE to enable interrupt FALSE to disable interrupt

Returns

None

bAHI_BrownOutStatus (JN5148 Only)

```
bool_t bAHI_BrownOutStatus(void);
```

Description

This function can be used to check whether the current supply voltage to the JN5148 device is above or below the brownout voltage setting (the default value or the value configured using the function **vAHI_BrownOutConfigure()**).

The function is useful when deciding on a suitable brownout voltage to configure.

There may be a delay of up to 30 µs before **bAHI_BrownOutStatus()** returns, if the brownout configuration has recently changed.

Parameters

None

Returns

TRUE if supply voltage is below brownout voltage

FALSE if supply voltage is above brownout voltage

bAHI_BrownOutEventResetStatus (JN5148 Only)

```
bool_t bAHI_BrownOutEventResetStatus(void);
```

Description

This function can be called following a JN5148 device reset to determine whether the reset event was caused by a brownout. This allows the application to then take any necessary action following a confirmed brownout.

Note that by default, a brownout will trigger a reset event. However, if **vAHI_BrownOutConfigure()** was called, the ‘reset on brownout’ option must have been explicitly enabled during this call.

Parameters

None

Returns

TRUE if brownout caused reset, FALSE otherwise

u32AHI_BrownOutPoll (JN5148 Only)

```
uint32 u32AHI_BrownOutPoll(void);
```

Description

This function can be used to poll for a brownout on the JN5148 device - that is, to check whether a brownout has occurred. The returned value will indicate whether the chip supply voltage has fallen below or risen above the brownout voltage (or both). Polling using this function clears the brownout status, so that a new and valid result will be obtained the next time the function is called.

Polling in this way is useful when brownout interrupts and 'reset on brownout' have been disabled through **vAHI_BrownOutConfigure()**. However, to successfully poll, brownout detection must still have been enabled through the latter function.

Parameters

None

Returns

32-bit value containing brownout status:

- Bit 24 is set (to '1') if the chip has come out of brownout - that is, an increasing supply voltage has crossed the brownout voltage from below. If the 32-bit return value is logically ANDed with the bitmask E_AHI_SYSCTRL_VFEM_MASK, a non-zero result indicates this brownout condition.
- Bit 25 is set (to '1') if the chip has gone into brownout - that is, a decreasing supply voltage has crossed the brownout voltage from above. If the 32-bit return value is logically ANDed with the bitmask E_AHI_SYSCTRL_VREM_MASK, a non-zero result indicates this brownout condition.

vAHI_SwReset

```
void vAHI_SwReset (void);
```

Description

This function generates an internal reset which completely re-starts the system through the full reset sequence.

Caution: *This reset has the same effect as pulling the external RESETN line low and is likely to result in the loss of the contents of on-chip RAM.*

Parameters

None

Returns

None

vAHI_DriveResetOut

```
void vAHI_DriveResetOut(uint8 u8Period);
```

Description

This function drives the ResetN line low for the specified period of time.

Note that one or more external devices may be connected to the ResetN line. Therefore, using this function to drive this line low may affect these external devices. For more information on the ResetN line and external devices, consult the datasheet for your wireless microcontroller.

Parameters

u8Period	Duration for which line will be driven low, in milliseconds

Returns

None

vAHI_ClearSystemEventStatus

```
void vAHI_ClearSystemEventStatus(uint32 u32BitMask);
```

Description

This function clears the specified System Controller interrupt sources. A bitmask indicating the interrupt sources to be cleared must be passed into the function.

Parameters

u32BitMask	Bitmask of the System Controller interrupt sources to be cleared. To clear an interrupt, the corresponding bit must be set to 1 - for bit numbers, refer to Table 2 on page 279

Returns

None

vAHI_SysCtrlRegisterCallback

```
void vAHI_SysCtrlRegisterCallback(
            PR_HWINT_APPCALLBACK prSysCtrlCallback);
```

Description

This function registers a user-defined callback function that will be called when a System Control interrupt is triggered. The source of this interrupt could be the wake timer, a comparator, a DIO event, a brownout event (JN5148 only), a pulse counter (JN5148 only) or the random number generator (JN5148 only).

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Note that the System Controller interrupt handler will clear the interrupt before invoking the callback function to deal with the interrupt.

Interrupt handling is described in Appendix A.

Parameters

prSysCtrlCallback	Pointer to callback function to be registered

Returns

None

4. Analogue Peripherals

This chapter describes the functions that are used to control the analogue peripherals of the Jennic wireless microcontrollers. These are the on-chip peripherals with analogue inputs or outputs, including an Analogue-to-Digital Converter (ADC), Digital-to-Analogue Converters (DACs) and comparators.

The analogue peripheral functions are divided into the following sections:

- Common analogue peripheral functions, described in Section 4.1
- ADC functions, described in Section 4.2
- DAC functions, described in Section 4.3
- Comparator functions, described in Section 4.4

Note: For guidance on using these functions in JN5148/JN5139 application code, refer to Chapter 4 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

4.1 Common Analogue Peripheral Functions

This section describes functions used to configure functionality shared by the on-chip analogue peripherals - the ADC, DACs and comparators.

The functions are listed below, along with their page references:

Function	Page
vAHI_ApConfigure	60
vAHI_ApSetBandGap	61
bAHI_APRegulatorEnabled	62
vAHI_APRegisterCallback	63

vAHI_ApConfigure

```
void vAHI_ApConfigure(bool_t bAPRegulator,
                      bool_t bIntEnable,
                      uint8 u8SampleSelect,
                      uint8 u8ClockDivRatio,
                      bool_t bRefSelect);
```

Description

This function configures common parameters for all on-chip analogue resources.

- The analogue peripheral regulator can be enabled - this dedicated power source minimises digital noise and is sourced from the analogue supply pin VDD1.
- Interrupts can be enabled that are generated after each ADC conversion.
- The clock frequency (derived from the chip's 16-MHz clock) is specified.
- The 'sampling interval' is specified as a number of clock periods.
- The source of the reference voltage, V_{ref}, is specified.

For the ADC, the input signal is integrated over *3 x sampling interval,* where *sampling interval* is defined as 2, 4, 6 or 8 clock cycles. For the ADC and DACs, the total conversion period (for a single value) is given by

[(3 x *sampling interval*) + 14] x *clock period*

Parameters

bAPRegulator	Enable/disable analogue peripheral regulator: E_AHI_AP_REGULATOR_ENABLE E_AHI_AP_REGULATOR_DISABLE
bIntEnable	Enable/disable interrupt when ADC conversion completes: E_AHI_AP_INT_ENABLE E_AHI_AP_INT_DISABLE
u8SampleSelect	Sampling interval in terms of divided clock periods (see below): E_AHI_AP_SAMPLE_2 (2 clock periods) E_AHI_AP_SAMPLE_4 (4 clock periods) E_AHI_AP_SAMPLE_6 (6 clock periods) E_AHI_AP_SAMPLE_8 (8 clock periods)
u8ClockDivRatio	Clock divisor (for 16-MHz clock): E_AHI_AP_CLOCKDIV_2MHZ (achieves 2 MHz) E_AHI_AP_CLOCKDIV_1MHZ (achieves 1 MHz) E_AHI_AP_CLOCKDIV_500KHZ (achieves 500 kHz) E_AHI_AP_CLOCKDIV_250KHZ (achieves 250 kHz) (500 kHz is recommended for ADC)
bRefSelect	Source of reference voltage, V_{ref}: E_AHI_AP_EXTREF (external from VREF pin) E_AHI_AP_INTREF (internal)

Returns

None

vAHI_ApSetBandGap

void vAHI_ApSetBandGap(bool_t *bBandGapEnable***);**

Description

This function allows the device's internal band-gap cell to be routed to the VREF pin, in order to provide internal reference voltage de-coupling.

Note that:

- Before calling **vAHI_ApSetBandGap()**, you must ensure that protocol power is enabled, by calling **vAHI_ProtocolPower()** if necessary, otherwise an exception will occur. Also, subsequently disabling protocol power will cause the band-gap cell setting to be lost.
- A call to **vAHI_ApSetBandGap()** is only valid if an internal source for V_{ref} has been selected through the function **vAHI_ApConfigure()**.

Caution: *Never call this function to enable the use of the internal band-gap cell after selecting an external source for V_{ref} through **vAHI_ApConfigure()**, otherwise damage to the device may result.*

Parameters

bBandGapEnable	Enable/disable routing of band-gap cell to VREF: E_AHI_AP_BANDGAP_ENABLE (enable routing) E_AHI_AP_BANDGAP_DISABLE (disable routing)

Returns

None

bAHI_APRegulatorEnabled

```
bool_t bAHI_APRegulatorEnabled(void);
```

Description

This function enquires whether the analogue peripheral regulator has powered up. The function should be called after enabling the regulator through **vAHI_ApConfigure()**. When the regulator is enabled, it will take a little time to start - this period is:

- 31.25 µs for the JN5148 and JN5139 devices
- 218.75 µs for the JN5121 device

Parameters

None

Returns

TRUE if powered up, FALSE if still waiting

vAHI_APRegisterCallback

```
void vAHI_APRegisterCallback(
                    PR_HWINT_APPCALLBACK prApCallback);
```

Description

This function registers a user-defined callback function that will be called when an Analogue Peripheral interrupt is triggered.

Note: Among the analogue peripherals, only the ADC generates Analogue Peripheral interrupts. The DACs do not generate interrupts and the comparators generate System Controller interrupts (see Chapter 3).

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A. Analogue Peripheral interrupt handling is further described in the *Integrated Peripherals API User Guide (JN-UG-3066)*.

Parameters

prApCallback	Pointer to callback function to be registered

Returns

None

4.2 ADC Functions

This section describes the functions that can be used to control the on-chip ADC (Analogue-to-Digital Converter). This is a 12-bit ADC that can be switched between 6 different sources (4 pins on the device, an on-chip temperature sensor and a voltage monitor). The ADC can be configured to perform a single conversion or convert continuously (until stopped). On the JN5148 device, it is also possible to operate the ADC in accumulation mode, in which a number of consecutive samples are added together for averaging.

The functions are listed below, along with their page references:

Function	**Page**
vAHI_AdcEnable	65
vAHI_AdcStartSample	66
vAHI_AdcStartAccumulateSamples (JN5148 Only)	67
bAHI_AdcPoll (JN5139/JN5148 Only)	68
u16AHI_AdcRead	69
vAHI_AdcDisable	70

Note: In order to use the ADC, the analogue peripheral regulator must first be enabled using the function **vAHI_ApConfigure()**. You must also check that the regulator has started, using the function **bAHI_APRegulatorEnabled()**.

vAHI_AdcEnable

```
void vAHI_AdcEnable(bool_t bContinuous,
                    bool_t bInputRange,
                    uint8 u8Source);
```

Description

This function configures and enables the ADC. Note that this function does not start the conversions (this is done using the function **vAHI_AdcStartSample()** or, in the case of accumulation mode on the JN5148 device, using the function **vAHI_AdcStartAccumulateSamples()**).

The function allows the ADC mode of operation to be set to one of:

- **Single-shot mode:** ADC will perform a single conversion and then stop (only valid if DACs are not enabled).
- **Continuous mode:** ADC will perform conversions repeatedly until stopped using the function **vAHI_AdcDisable()**.

If using the ADC in accumulation mode (JN5148 only), the mode set here is ignored.

The function also allows the input source for the ADC to be selected as one of four pins, the on-chip temperature sensor or the internal voltage monitor. The voltage range for the analogue input to the ADC can also be selected as 0-V_{ref} or 0-2V_{ref}.

Note that:

- The source of V_{ref} is defined using **vAHI_ApConfigure()**.
- The internal voltage monitor measures the voltage on the pin VDD1.

Before enabling the ADC, the analogue peripheral regulator must have been enabled using the function **vAHI_ApConfigure()**. You must also check that the regulator has started, using the function **bAHI_APRegulatorEnabled()**.

Parameters

bContinuous	Conversion mode of ADC: E_AHI_ADC_CONTINUOUS (continous mode) E_AHI_ADC_SINGLE_SHOT (single-shot mode)
bInputRange	Input voltage range: E_AHI_AP_INPUT_RANGE_1 (0 to V_{ref}) E_AHI_AP_INPUT_RANGE_2 (0 to 2V_{ref})
u8Source	Source for conversions: E_AHI_ADC_SRC_ADC_1 (ADC1 input) E_AHI_ADC_SRC_ADC_2 (ADC2 input) E_AHI_ADC_SRC_ADC_3 (ADC3 input) E_AHI_ADC_SRC_ADC_4 (ADC4 input) E_AHI_ADC_SRC_TEMP (on-chip temperature sensor) E_AHI_ADC_SRC_VOLT (internal voltage monitor)

Returns

None

vAHI_AdcStartSample

void vAHI_AdcStartSample(void);

Description

This function starts the ADC sampling in single-shot or continuous mode, depending on which mode has been configured using **vAHI_AdcEnable()**.

If analogue peripheral interrupts have been enabled in **vAHI_ApConfigure()**, an interrupt will be triggered when a result becomes available. Alternatively, if interrupts are disabled, you can use **bAHI_AdcPoll()** to check for a result. Once a conversion result becomes available, it should be read with **u16AHI_AdcRead()**.

Once sampling has been started in continuous mode, it can be stopped at any time using the function **vAHI_AdcDisable()**.

Note: On the JN5148 device, if you wish to use the ADC in accumulation mode, start sampling using the function **vAHI_AdcStartAccumulateSamples()** instead.

Parameters

None

Returns

None

vAHI_AdcStartAccumulateSamples (JN5148 Only)

```
void vAHI_AdcStartAccumulateSamples(
                              uint8 u8AccSamples);
```

Description

This function starts the ADC sampling in accumulation mode on the JN5148 device, which allows a specified number of consecutive samples to be added together to facilitate the averaging of output samples. Note that before calling this function, the ADC must be configured and enabled using **vAHI_AdcEnable()**.

In accumulation mode, the output will become available after the specified number of consecutive conversions (2, 4, 8 or 16), where this output is the sum of these conversion results. Conversion will then stop. The cumulative result can be obtained using the function **u16AHI_AdcRead()**, but the application must then perform the averaging calculation itself (by dividing the result by the appropriate number of samples).

If analogue peripheral interrupts have been enabled in **vAHI_ApConfigure()**, an interrupt will be triggered when the accumulated result becomes available. Alternatively, if interrupts are disabled, you can use the function **bAHI_AdcPoll()** to check whether the conversions have completed.

In this mode, conversion can be stopped at any time using the function **vAHI_AdcDisable()**.

Parameters

u8AccSamples	Number of samples to add together: E_AHI_ADC_ACC_SAMPLE_2 (2 samples) E_AHI_ADC_ACC_SAMPLE_4 (4 samples) E_AHI_ADC_ACC_SAMPLE_8 (8 samples) E_AHI_ADC_ACC_SAMPLE_16 (16 samples)

Returns

None

bAHI_AdcPoll (JN5139/JN5148 Only)

```
bool_t bAHI_AdcPoll(void);
```

Description

This function can be used on the JN5148/JN5139 device, when the ADC is operating in single-shot mode, continuous mode or accumulation mode (JN5148 only), to check whether the ADC is still busy performing a conversion:

- In single-shot mode, the poll result indicates whether the sample has been taken and is ready to be read.
- In continuous mode, the poll result indicates whether a new sample is ready to be read.
- In accumulation mode on the JN5148 device, the poll result indicates whether the final sample for the accumulation has been taken.

You may wish to call this function before attempting to read the conversion result using **u16AHI_AdcRead()**, particularly if you are not using the analogue peripheral interrupts.

Parameters

None

Returns

TRUE if ADC is busy, FALSE if conversion complete

u16AHI_AdcRead

```
uint16 u16AHI_AdcRead(void);
```

Description

This function reads the most recent ADC conversion result.

- If sampling was started using the function **vAHI_AdcStartSample()**, the most recent ADC conversion will be returned.
- If sampling on the JN5148 device was started using the function **vAHI_AdcStartAccumulateSamples()**, the last accumulated conversion result will be returned.

If analogue peripheral interrupts have been enabled in **vAHI_ApConfigure()**, you must call this read function from a callback function invoked when an interrupt has been generated to indicate that an ADC result is ready (this user-defined callback function is registered using the function **vAHI_APRegisterCallback()**). Alternatively, if interrupts have not been enabled, before calling the read function, you must first check whether a result is ready using the function **bAHI_AdcPoll()**.

Parameters

None

Returns

Most recent ADC conversion result (the result is contained in the least significant 12 bits of the 16-bit returned value) or, if in accumulation mode, the most recent accumulated conversion result (here, all 16 bits are relevant)

vAHI_AdcDisable

```
void vAHI_AdcDisable(void);
```

Description

This function disables the ADC. It can be used to stop the ADC when operating in continuous mode or accumulation mode (the latter mode on JN5148 only).

Parameters

None

Returns

None

4.3 DAC Functions

This section describes the functions that can be used to control the on-chip DACs (Digital-to-Analogue Converters). The Jennic wireless microcontrollers feature two DACs, denoted DAC1 and DAC2. On the JN5148 device, 12-bit DACs are used, while on the JN5139 and JN5121 devices, 11-bit DACs are used. The outputs from these DACs go to dedicated pins on the chip.

The functions are listed below, along with their page references:

Function	**Page**
vAHI_DacEnable	72
vAHI_DacOutput	73
bAHI_DacPoll	74
vAHI_DacDisable	75

Note 1: In order to use a DAC, the analogue peripheral regulator must first be enabled using the function **vAHI_ApConfigure()**. You must also check that the regulator has started, using the function **bAHI_APRegulatorEnabled()**.

Note 2: On the JN5139 and JN5121 devices, only one DAC can be enabled at any one time. If both DACs are to be used concurrently, they can be multiplexed.

Note 3: When a DAC is enabled, the ADC cannot be used in single-shot mode but can be used in continuous mode.

vAHI_DacEnable

```
void vAHI_DacEnable(uint8 u8Dac,
                    bool_t bOutputRange,
                    bool_t bRetainOutput,
                    uint16 u16InitialVal);
```

Description

This function configures and enables the specified DAC (DAC1 or DAC2). Note that:

- On JN5139 and JN5121, only one of the DACs can be enabled at any one time. If both DACs are to be used concurrently, they can be multiplexed.
- The voltage range for the analogue output can be specified as 0-V_{ref} or 0-2V_{ref}.
- The source of V_{ref} is defined using **vAHI_ApConfigure()**.
- The first value to be converted is specified through this function (for JN5148 only). Subsequent values must be specified through **vAHI_DacOutput()**.

Caution: *The parameter u16InitialVal is not used by the JN5139 and JN5121 devices. To set the initial value to be converted (and all subsequent values), use the function* ***vAHI_DacOutput()****.*

Before enabling the DAC, the analogue peripheral regulator must have been enabled using the function **vAHI_ApConfigure()**. You must also check that the regulator has started, using the function **bAHI_APRegulatorEnabled()**.

When a DAC is enabled, the ADC cannot be used in single-shot mode but can be used in continuous mode.

Parameters

u8Dac	DAC to configure and enable: E_AHI_AP_DAC_1 (DAC1) E_AHI_AP_DAC_2 (DAC2)
bOutputRange	Output voltage range: E_AHI_AP_INPUT_RANGE_1 (0 to V_{ref}) E_AHI_AP_INPUT_RANGE_2 (0 to 2V_{ref})
bRetainOutput	Unused - set to 0 (FALSE)
u16InitialVal	Initial value to be converted - only the 12 least significant bits will be used (this parameter is not valid for the JN5139 and JN5121 devices - see above)

Returns

None

vAHI_DacOutput

```
void vAHI_DacOutput(uint8 u8Dac,
                    uint16 u16Value);
```

Description

This function allows the next value for conversion by the specified DAC to be set. This value will be used for all subsequent conversions until the function is called again with a new value.

Although a 16-bit value must be specified in this function:

- For the JN5148 device, only the 12 least significant bits will be used, since the chip features 12-bit DACs
- For the JN5139 and JN5121 devices, only the 11 least significant bits will be used, since the chip features 11-bit DACs

Parameters

u8Dac	DAC to which value will be submitted: E_AHI_AP_DAC_1 (DAC1) E_AHI_AP_DAC_2 (DAC2)
u16Value	Value to be converted - only the 11 or 12 least significant bits will be used (see above)

Returns

None

bAHI_DacPoll

```
bool_t bAHI_DacPoll(void);
```

Description

This function can be used to check whether the enabled DAC is busy performing a conversion. A short delay (of approximately 2 µs) is included after polling and determining whether the DAC has completed, in order to prevent lock-ups when further calls are made to the DAC.

Parameters

None

Returns

TRUE if DAC is busy, FALSE if conversion complete

vAHI_DacDisable

```
void vAHI_DacDisable(uint8 u8Dac);
```

Description

This function stops and powers down the specified DAC.

Note that on the JN5139 and JN5121 devices, only one of the two DACs can be used at any one time. If both DACs are to be used concurrently, they can be multiplexed.

Parameters

u8Dac	DAC to disable: E_AHI_AP_DAC_1 (DAC1) E_AHI_AP_DAC_2 (DAC2)

Returns

None

4.4 Comparator Functions

This section describes the functions that can be used to control the on-chip comparators:

- On the JN5139 and JN5148 devices, there are two comparators (1 and 2)
- On the JN5121 device, there is one comparator

A comparator compares its signal input with a reference input, and can be programmed to provide an interrupt when the difference between its inputs changes sense. It can also be used to wake the chip from sleep. The inputs to the comparator use dedicated pins on the chip. The signal input is provided on the comparator '+' pin and the reference input is provided on the comparator '-' pin, by the DAC output or by the internal reference voltage V_{ref}.

Note: If the comparator is to be used to wake the device from sleep mode then only the comparator '+' and '-' pins can be used. The internal reference voltage cannot be used and neither can the DAC output (as the DACs are switched off when the device enters sleep mode).

Note: The analogue peripheral regulator must be enabled while configuring a comparator, although it can be disabled once configuration is complete.

The following comparator functions are available for the JN5121 device:

Function	**Page**
vAHI_CompEnable (JN5121 Only)	77
vAHI_CompDisable (JN5121 Only)	78
vAHI_CompIntEnable (JN5121 Only)	79
u8AHI_CompStatus (JN5121 Only)	80
u8AHI_CompWakeStatus (JN5121 Only)	81

The following comparator functions are available for the JN5139 and JN5148 devices:

Function	**Page**
vAHI_ComparatorEnable (JN5139/JN5148 Only)	82
vAHI_ComparatorDisable (JN5139/JN5148 Only)	83
vAHI_ComparatorLowPowerMode (JN5139/JN5148 Only)	84
vAHI_ComparatorIntEnable (JN5139/JN5148 Only)	85
u8AHI_ComparatorStatus (JN5139/JN5148 Only)	86
u8AHI_ComparatorWakeStatus (JN5139/JN5148 Only)	87

vAHI_CompEnable (JN5121 Only)

```
void vAHI_CompEnable(uint8 u8Hysteresis,
                     uint8 u8SignalSelect);
```

Description

This function configures and enables the comparator on the JN5121 device. The reference signal and hysteresis setting must be specified.

The hysteresis voltage selected should be greater than:

- the noise level in the input signal on the comparator '+' pin, if comparing the signal on this pin with the internal reference voltage or DAC output
- the differential noise between the signals on the comparator '+' and '-' pins, if comparing the signals on these two pins

Once enabled using this function, the comparator can be disabled using the function **vAHI_CompDisable()**.

Parameters

u8Hysteresis	Hysteresis setting (controllable from 0 to 20 mV) E_AHI_COMP_HYSTERESIS_0MV (0 mV) E_AHI_COMP_HYSTERESIS_5MV (5 mV) E_AHI_COMP_HYSTERESIS_10MV (10 mV) E_AHI_COMP_HYSTERESIS_20MV (20 mV)
u8SignalSelect	Reference signal to compare with input signal on comparator '+' pin: E_AHI_COMP_SEL_EXT (comparator '-' pin) E_AHI_COMP_SEL_DAC (DAC2 output) E_AHI_COMP_SEL_BANDGAP (fixed at V_{ref})

Returns

None

vAHI_CompDisable (JN5121 Only)

```
void vAHI_CompDisable(void);
```

Description

This function disables the comparator on the JN5121 device.

Parameters

None

Returns

None

vAHI_CompIntEnable (JN5121 Only)

```
void vAHI_CompIntEnable(bool_t bIntEnable,
                        bool_t bRisingNotFalling);
```

Description

This function enables interrupts for the comparator on the JN5121 device. An interrupt can be used to wake the device from sleep or as a normal interrupt.

If enabled, an interrupt is generated on one of the following conditions (which must be configured):

- The input signal rises above the reference signal (plus hysteresis level, if non-zero)
- The input signal falls below the reference signal (minus hysteresis level, if non-zero)

Comparator interrupts are handled by the System Controller callback function, registered using the function **vAHI_SysCtrlRegisterCallback()**.

Parameters

bIntEnable	Enable/disable interrupts: TRUE to enable interrupts FALSE to disable interrupts
bRisingNotFalling	Triggering condition for interrupt: TRUE for interrupt when input signal rises above reference FALSE for interrupt when input signal falls below reference

Returns

None

u8AHI_CompStatus (JN5121 Only)

```
uint8 u8AHI_CompStatus(void);
```

Description

This function obtains the status of the comparator on the JN5121 device.

The result is interpreted as follows:

- **0** indicates that the input signal is lower than the reference signal
- **1** indicates that the input signal is higher than the reference signal

Parameters

None

Returns

Value indicating status of comparator (see above)

u8AHI_CompWakeStatus (JN5121 Only)

```
uint8 u8AHI_CompWakeStatus(void);
```

Description

This function returns the wake-up interrupt status of the comparator on the JN5121 device. The value is cleared after reading.

The result is interpreted as follows:

- **Zero** indicates that a wake-up interrupt has not occurred
- **Non-zero** value indicates that a wake-up interrupt has occurred

Note: If you wish to use this function to check whether the comparator caused a wake-up event, you must call it before **u32AHI_Init()**. Alternatively, you can determine the wake source as part of your System Controller callback function. For more information, refer to Appendix A.

Parameters

None

Returns

Value indicating wake-up interrupt status of comparator (see above)

vAHI_ComparatorEnable (JN5139/JN5148 Only)

```
void vAHI_ComparatorEnable(uint8 u8Comparator,
                           uint8 u8Hysteresis,
                           uint8 u8SignalSelect);
```

Description

This function configures and enables the specified comparator on the JN5139/ JN5148 device. The reference signal and hysteresis setting must be specified.

The hysteresis voltage selected should be greater than:

- the noise level in the input signal on the comparator '+' pin, if comparing the signal on this pin with the internal reference voltage or DAC output
- the differential noise between the signals on the comparator ‘+’ and ‘-’ pins, if comparing the signals on these two pins

Note that the same hysteresis setting is used for both comparators, so if this function is called several times for different comparators, only the hysteresis value from the final call will be used.

Note: This function puts the comparator in low-power mode in which the comparator draws 1.2 µA of current, compared with 70 µA when operating in standard-power mode. If you wish to use the comparators in standard-power mode, you must disable low-power mode using the function **vAHI_ComparatorLowPowerMode()**.

Once enabled using this function, the comparator can be disabled using the function **vAHI_ComparatorDisable()**.

Parameters

u8Comparator	Identity of comparator: E_AHI_AP_COMPARATOR_1 E_AHI_AP_COMPARATOR_2
u8Hysteresis	Hysteresis setting (controllable from 0 to 40 mV) E_AHI_COMP_HYSTERESIS_0MV (0 mV) E_AHI_COMP_HYSTERESIS_10MV (10 mV) E_AHI_COMP_HYSTERESIS_20MV (20 mV) E_AHI_COMP_HYSTERESIS_40MV (40 mV)
u8SignalSelect	Reference signal to compare with input signal on comparator '+' pin: E_AHI_COMP_SEL_EXT (comparator '-' pin) E_AHI_COMP_SEL_DAC (related DAC output) E_AHI_COMP_SEL_BANDGAP (fixed at V_{ref})

Returns

None

vAHI_ComparatorDisable (JN5139/JN5148 Only)

```
void vAHI_ComparatorDisable(uint8 u8Comparator);
```

Description

This function disables the specified comparator on the JN5139/JN5148 device.

Parameters

u8Comparator	Identity of comparator: E_AHI_AP_COMPARATOR_1 E_AHI_AP_COMPARATOR_2

Returns

None

vAHI_ComparatorLowPowerMode (JN5139/JN5148 Only)

```
void vAHI_ComparatorLowPowerMode(
                                bool_t bLowPowerEnable);
```

Description

This function can be used to enable or disable low-power mode on the comparators of the JN5139/JN5148 device. The function affects both comparators together.

In low-power mode, a comparator draws 1.2 µA of current, compared with 70 µA when operating in standard-power mode. Low-power mode can be used while the device is sleeping, to minimise power consumption, but is also ideal for energy harvesting (while awake).

When a comparator is enabled using **vAHI_ComparatorEnable()**, it is put into low-power mode by default. Therefore, to use the comparators in standard-power mode, you must call **vAHI_ComparatorLowPowerMode()** to disable low-power mode.

Parameters

bLowPowerEnable	Enable/disable low-power mode: TRUE - enable FALSE - disable

Returns

None

vAHI_ComparatorIntEnable (JN5139/JN5148 Only)

```
void vAHI_ComparatorIntEnable(uint8 u8Comparator,
                              bool_t bIntEnable,
                              bool_t bRisingNotFalling);
```

Description

This function enables interrupts for the specified comparator on the JN5139/JN5148 device. An interrupt can be used to wake the device from sleep or as a normal interrupt.

If enabled, an interrupt is generated on one of the following conditions (which must be configured):

- The input signal rises above the reference signal (plus hysteresis level, if non-zero)
- The input signal falls below the reference signal (minus hysteresis level, if non-zero)

Comparator interrupts are handled by the System Controller callback function, registered using the function **vAHI_SysCtrlRegisterCallback()**.

Parameters

u8Comparator	Identity of comparator: E_AHI_AP_COMPARATOR_1 E_AHI_AP_COMPARATOR_2
bIntEnable	Enable/disable interrupts: TRUE to enable interrupts FALSE to disable interrupts
bRisingNotFalling	Triggering condition for interrupt: TRUE for interrupt when input signal rises above reference FALSE for interrupt when input signal falls below reference

Returns

None

u8AHI_ComparatorStatus (JN5139/JN5148 Only)

```
uint8 u8AHI_ComparatorStatus(void);
```

Description

This function obtains the status of the comparators on the JN5139/JN5148 device.

To obtain the status of an individual comparator, the returned value must be bitwise ANDed with the mask E_AHI_AP_COMPARATOR_MASK_x, where x is 1 for Comparator 1 and 2 for Comparator 2.

The result for an individual comparator is interpreted as follows:

- **0** indicates that the input signal is lower than the reference signal
- **1** indicates that the input signal is higher than the reference signal

Parameters

None

Returns

Value containing the status of both comparators (see above)

u8AHI_ComparatorWakeStatus (JN5139/JN5148 Only)

uint8 u8AHI_ComparatorWakeStatus(void);

Description

This function returns the wake-up interrupt status of the comparators on the JN5139/JN5148 device. The value is cleared after reading.

To obtain the wake-up interrupt status of an individual comparator, the returned value must be bitwise ANDed with the mask E_AHI_AP_COMPARATOR_MASK_x, where x is 1 for Comparator 1 and 2 for Comparator 2.

The result for an individual comparator is interpreted as follows:

- **Zero** indicates that a wake-up interrupt has not occurred
- **Non-zero** value indicates that a wake-up interrupt has occurred

Note: If you wish to use this function to check whether a comparator caused a wake-up event, you must call it before **u32AHI_Init()**. Alternatively, you can determine the wake source as part of your System Controller callback function. For more information, refer to Appendix A.

Parameters

None

Returns

Value containing wake-up interrupt status of both comparators (see above)

5. DIOs

This chapter describes the functions that can be used to control the digital input/output lines, referred to as DIOs. The Jennic wireless microcontrollers have 21 DIO lines, numbered 0 to 20, where each DIO can be individually configured. However, the pins for the DIO lines are shared with other peripherals (see list below) and are not available when those peripherals are enabled:

- UARTs
- Timers
- Serial Interface (2-wire)
- Serial Peripheral Interface
- Intelligent Peripheral Interface
- Antenna Diversity
- Pulse Counters [JN5148 only]
- Digital Audio Interface (DAI) [JN5148 only]

For details of the shared pins, refer to the data sheet for your wireless microcontroller.

In addition to normal operation, when configured as inputs, the DIOs can be used to generate interrupts and wake the device from sleep.

The DIO functions are listed below, along with their page references:

Function	**Page**
vAHI_DioSetDirection	90
vAHI_DioSetOutput	91
u32AHI_DioReadInput	92
vAHI_DioSetPullup	93
vAHI_DioSetByte (JN5148 Only)	94
u8AHI_DioReadByte (JN5148 Only)	95
vAHI_DioInterruptEnable	96
vAHI_DioInterruptEdge	97
u32AHI_DioInterruptStatus	98
vAHI_DioWakeEnable	99
vAHI_DioWakeEdge	100
u32AHI_DioWakeStatus	101

Note: For guidance on using these functions in JN5148/JN5139 application code, refer to Chapter 5 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

vAHI_DioSetDirection

```
void vAHI_DioSetDirection(uint32 u32Inputs,
                          uint32 u32Outputs);
```

Description

This function sets the direction for the DIO pins individually as either input or output (note that they are set as inputs, by default, on reset). This is done through two bitmaps for inputs and outputs, *u32Inputs* and *u32Outputs* respectively. In these values, each of bits 0 to 20 represents a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20 (bits 21-31 are ignored). Setting a bit in one of these bitmaps configures the corresponding DIO as an input or output, depending on the bitmap.

Note that:

- Not all DIO pins must be defined (in other words, *u32Inputs* logical ORed with *u32Outputs* does not need to produce all zeros for bits 0-20).
- Any DIO pins that are not defined by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Inputs* and *u32Outputs*, it will default to becoming an input.
- If a DIO is assigned to another peripheral which is enabled, this function call will not immediately affect the relevant pin. However, the DIO setting specified by this function will take effect if/when the relevant peripheral is subsequently disabled.
- This function does not change the DIO pull-up status - this must be done separately using **vAHI_DioSetPullup()**.

Parameters

u32Inputs	Bitmap of inputs - a bit set means that the corresponding DIO pin will become an input
u32Outputs	Bitmap of outputs - a bit set means that the corresponding DIO pin will become an output

Returns

None

vAHI_DioSetOutput

```
void vAHI_DioSetOutput(uint32 u32On, uint32 u32Off);
```

Description

This function sets individual DIO outputs on or off, driving an output high or low, respectively. This is done through two bitmaps for on-pins and off-pins, *u32On* and *u32Off* respectively. In these values, each of bits 0 to 20 represents a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20 (bits 21-31 are ignored). Setting a bit in one of these bitmaps configures the corresponding DIO output as on or off, depending on the bitmap.

Note that:

- Not all DIO pins must be defined (in other words, *u32On* logical ORed with *u32Off* does not need to produce all zeros for bits 0-20).
- Any DIO pins that are not defined by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32On* and *u32Off*, the DIO pin will default to off.
- This call has no effect on DIO pins that are not defined as outputs (see **vAHI_DioSetDirection()**), until a time when they are re-configured as outputs.
- If a DIO is assigned to another peripheral which is enabled, this function call will not affect the relevant DIO, until a time when the relevant peripheral is disabled.

Parameters

u32On	Bitmap of on-pins - a bit set means that the corresponding DIO pin will be set to on
u32Off	Bitmap of off-pins - a bit set means that the corresponding DIO pin will be set to off

Returns

None

u32AHI_DioReadInput

```
uint32 u32AHI_DioReadInput (void);
```

Description

This function returns the value of each of the DIO pins (irrespective of whether the pins are used as inputs, as outputs or by other enabled peripherals).

Parameters

None

Returns

Bitmap:

Each of bits 0-20 corresponds to a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20. The bit is set to 1 if the pin is high or to 0 if the pin is low. Bits 21-31 are always 0.

vAHI_DioSetPullup

void vAHI_DioSetPullup(uint32 *u32On*, **uint32** *u32Off***);**

Description

This function sets the pull-ups on individual DIO pins as on or off. A pull-up can be set irrespective of whether the pin is an input or output. This is done through two bitmaps for 'pull-ups on' and 'pull-ups off', *u32On* and *u32Off* respectively. In these values, each of bits 0 to 20 represents a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20 (bits 21-31 are ignored).

Note that:

- By default, the pull-ups are enabled (on) at power-up.
- Not all DIO pull-ups must be set (in other words, *u32On* logical ORed with *u32Off* does not need to produce all zeros for bits 0-20).
- Any DIO pull-ups that are not set by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32On* and *u32Off*, the corresponding DIO pull-up will default to off.
- If a DIO is assigned to another peripheral which is enabled, this function call will still apply to the relevant pin, except in the case of a DIO connected to an external 32-kHz crystal (JN5148 only).

Parameters

u32On	Bitmap of 'pull-ups on' - a bit set means that the corresponding pull-up will be disabled
u32Off	Bitmap of 'pull-ups off' - a bit set means that the corresponding pull-up will be disabled

Returns

None

vAHI_DioSetByte (JN5148 Only)

```
void vAHI_DioSetByte(bool_t bDIOSelect, uint8 u8DataByte);
```

Description

This function can be used to output a byte on either DIO0-7 or DIO8-15, where bit 0 or 8 is used for the least significant bit of the byte.

Before calling this function, the relevant DIOs must be configured as outputs using the function **vAHI_DioSetDirection()**.

Parameters

bDIOSelect	The set of DIO lines on which to output the byte: FALSE selects DIO0-7 TRUE selects DIO8-15
u8DataByte	The byte to output on the DIO pins

Returns

None

u8AHI_DioReadByte (JN5148 Only)

```
uint8 u8AHI_DioReadByte(bool_t bDIOSelect);
```

Description

This function can be used to read a byte input on either DIO0-7 or DIO8-15, where bit 0 or 8 is used for the least significant bit of the byte.

Before calling this function, the relevant DIOs must be configured as inputs using the function **vAHI_DioSetDirection()**.

Parameters

bDIOSelect	The set of DIO lines on which to read the input byte: FALSE selects DIO0-7 TRUE selects DIO8-15

Returns

The byte read from DIO0-7 or DIO8-15

vAHI_DioInterruptEnable

```
void vAHI_DioInterruptEnable(uint32 u32Enable,
                             uint32 u32Disable);
```

Description

This function enables/disables interrupts on the DIO pins - that is, whether the signal on a DIO pin will generate an interrupt. This is done through two bitmaps for ‘interrupts enabled’ and ‘interrupts disabled’, *u32Enable* and *u32Disable* respectively. In these values, each of bits 0 to 20 represents a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20 (bits 21-31 are ignored). Setting a bit in one of these bitmaps enables/disables interrupts on the corresponding DIO, depending on the bitmap (by default, all DIO interrupts are disabled).

Note that:

- Not all DIO interrupts must be defined (in other words, *u32Enable* logical ORed with *u32Disable* does not need to produce all zeros for bits 0-20).
- Any DIO interrupts that are not defined by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Enable* and *u32Disable*, the corresponding DIO interrupt will default to disabled.
- This call has no effect on DIO pins that are not defined as inputs (see **vAHI_DioSetDirection()**).
- DIOs assigned to enabled JN51xx peripherals are affected by this function.
- The DIO interrupt settings made with this function are retained during sleep.

The signal edge on which each DIO interrupt is generated can be configured using the function **vAHI_DioInterruptEdge()** (the default is ‘rising edge’).

DIO interrupts are handled by the System Controller callback function, registered using the function **vAHI_SysCtrlRegisterCallback()**.

Caution: *This function has the same effect as* ***vAHI_DioWakeEnable()*** *- both functions access the same JN51xx register bits. Therefore, do not allow the two functions to conflict in your code.*

Parameters

u32Enable	Bitmap of DIO interrupts to enable - a bit set means that interrupts on the corresponding DIO will be enabled
u32Disable	Bitmap of DIO interrupts to disable - a bit set means that interrupts on the corresponding DIO will be disabled

Returns

None

vAHI_DioInterruptEdge

```
void vAHI_DioInterruptEdge(uint32 u32Rising,
                           uint32 u32Falling);
```

Description

This function configures enabled DIO interrupts by controlling whether individual DIOs will generate interrupts on a rising or falling edge of the DIO signal. This is done through two bitmaps for 'rising edge' and 'falling edge', *u32Rising* and *u32Falling* respectively. In these values, each of bits 0 to 20 represents a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20 (bits 21-31 are ignored). Setting a bit in one of these bitmaps configures interrupts on the corresponding DIO to occur on a rising or falling edge, depending on the bitmap (by default, all DIO interrupts are 'rising edge').

Note that:

- Not all DIO interrupts must be configured (in other words, *u32Rising* logical ORed with *u32Falling* does not need to produce all zeros for bits 0-20).
- Any DIO interrupts that are not configured by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Rising* and *u32Falling*, the corresponding DIO interrupt will default to 'rising edge'.
- This call has no effect on DIO pins that are not defined as inputs (see **vAHI_DioSetDirection()**).
- DIOs assigned to enabled JN51xx peripherals are affected by this function.
- The DIO interrupt settings made with this function are retained during sleep.

The DIO interrupts can be individually enable/disabled using the function **vAHI_DioInterruptEnable()**.

***Caution:** This function has the same effect as **vAHI_DioWakeEdge()** - both functions access the same JN51xx register bits. Therefore, do not allow the two functions to conflict in your code.*

Parameters

u32Rising	Bitmap of DIO interrupts to configure - a bit set means that interrupts on the corresponding DIO will be generated on a rising edge
u32Falling	Bitmap of DIO interrupts to configure - a bit set means that interrupts on the corresponding DIO will be generated on a falling edge

Returns

None

u32AHI_DioInterruptStatus

```
uint32 u32AHI_DioInterruptStatus(void);
```

Description

This function obtains the interrupt status of all the DIO pins. It is used to poll the DIO interrupt status when DIO interrupts are disabled (and therefore not generated).

Tip: If you wish to generate DIO interrupts instead of using this function to poll, you must enable DIO interrupts using **vAHI_DioInterruptEnable()** and incorporate DIO interrupt handling in the System Controller callback function registered using **vAHI_SysCtrlRegisterCallback()**.

The returned value is a bitmap in which a bit is set if an interrupt has occurred on the corresponding DIO pin (see below). In addition, this bitmap reports other DIO events that have occurred. After reading, the interrupt status and any other reported DIO events are cleared.

The results are valid irrespective of whether the pins are used as inputs, as outputs or by other enabled peripherals. They are also valid immediately following sleep.

Note: This function has the same effect as **vAHI_DioWakeStatus()** - both functions access the same JN51xx register bits.

Parameters

None

Returns

Bitmap:

Each of bits 0-20 corresponds to a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20. The bit is set to 1 if the corresponding DIO interrupt has occurred or to 0 if it has not occurred. Bits 21-31 are always 0.

vAHI_DioWakeEnable

void vAHI_DioWakeEnable(uint32 *u32Enable*,
uint32 *u32Disable***);**

Description

This function enables/disables wake interrupts on the DIO pins - that is, whether activity on a DIO input will be able to wake the device from sleep or doze mode. This is done through two bitmaps for 'wake enabled' and 'wake disabled', *u32Enable* and *u32Disable* respectively. In these values, each of bits 0 to 20 represents a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20 (bits 21-31 are ignored). Setting a bit in one of these bitmaps enables/disables wake interrupts on the corresponding DIO, depending on the bitmap.

Note that:

- Not all DIO wake interrupts must be defined (in other words, *u32Enable* logical ORed with *u32Disable* does not need to produce all zeros for bits 0-20).
- Any DIO wake interrupts that are not defined by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Enable* and *u32Disable*, the corresponding DIO wake interrupt will default to disabled.
- This call has no effect on DIO pins that are not defined as inputs (see **vAHI_DioSetDirection()**).
- DIOs assigned to enabled JN51xx peripherals are affected by this function.
- The DIO wake interrupt settings made with this function are retained during sleep.

The signal edge on which each DIO wake interrupt is generated can be configured using the function **vAHI_DioWakeEdge()** (the default is 'rising edge').

DIO wake interrupts are handled by the System Controller callback function, registered using the function **vAHI_SysCtrlRegisterCallback()**.

Caution: *This function has the same effect as* ***vAHI_DioInterruptEnable()*** *- both functions access the same JN51xx register bits. Therefore, do not allow the two functions to conflict in your code.*

Parameters

u32Enable	Bitmap of DIO wake interrupts to enable - a bit set means that wake interrupts on the corresponding DIO will be enabled
u32Disable	Bitmap of DIO wake interrupts to disable - a bit set means that wake interrupts on the corresponding DIO will be disabled

Returns

None

vAHI_DioWakeEdge

```
void vAHI_DioWakeEdge(uint32 u32Rising,
                      uint32 u32Falling);
```

Description

This function configures enabled DIO wake interrupts by controlling whether individual DIOs will generate wake interrupts on a rising or falling edge of the DIO input. This is done through two bitmaps for ‘rising edge’ and ‘falling edge’, *u32Rising* and *u32Falling* respectively. In these values, each of bits 0 to 20 represents a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20 (bits 21-31 are ignored). Setting a bit in one of these bitmaps configures wake interrupts on the corresponding DIO to occur on a rising or falling edge, depending on the bitmap (by default, all DIO wake interrupts are ‘rising edge’).

Note that:

- Not all DIO wake interrupts must be configured (in other words, *u32Rising* logical ORed with *u32Falling* does not need to produce all zeros for bits 0-20).
- Any DIO wake interrupts that are not configured by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Rising* and *u32Falling*, the corresponding DIO wake interrupt will default to ‘rising edge’.
- This call has no effect on DIO pins that are not defined as inputs (see **vAHI_DioSetDirection()**).
- DIOs assigned to enabled JN51xx peripherals are affected by this function.
- The DIO wake interrupt settings made with this function are retained during sleep.

The DIO wake interrupts can be individually enable/disabled using the function **vAHI_DioWakeEnable()**.

> ***Caution:*** *This function has the same effect as* ***vAHI_DioInterruptEdge()*** *- both functions access the same JN51xx register bits. Therefore, do not allow the two functions to conflict in your code.*

Parameters

u32Rising	Bitmap of DIO wake interrupts to configure - a bit set means that wake interrupts on the corresponding DIO will be generated on a rising edge
u32Falling	Bitmap of DIO wake interrupts to configure - a bit set means that wake interrupts on the corresponding DIO will be generated on a falling edge

Returns

None

u32AHI_DioWakeStatus

```
uint32 u32AHI_DioWakeStatus(void);
```

Description

This function returns the wake status of all the DIO input pins - that is, whether the DIO pins were used to wake the device from sleep.

Note: If you wish to use this function to check whether a DIO caused a wake-up event, you must call it before **u32AHI_Init()**. Alternatively, you can determine the wake source as part of your System Controller callback function. For more information, refer to Appendix A.

The returned value is a bitmap in which a bit is set if a wake interrupt has occurred on the corresponding DIO input pin (see below). In addition, this bitmap reports other DIO events that have occurred. After reading, the wake status and any other reported DIO events are cleared.

The results are not valid for DIO pins that are configured as outputs or assigned to other enabled peripherals.

Note: This function has the same effect as **vAHI_DioInterruptStatus()** - both functions access the same JN51xx register bits.

Parameters

None

Returns

Bitmap:

Each of bits 0-20 corresponds to a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20. The bit is set to 1 if the corresponding DIO wake interrupt has occurred or to 0 if it has not occurred. Bits 21-31 are always 0.

6. UARTs

This chapter details the functions for controlling the on-chip UARTs (Universal Asynchronous Receiver Transmitters). The Jennic wireless microcontrollers have two 16550-compatible UARTs, denoted UART0 and UART1, which can be independently enabled.

Each UART uses four pins (shared with the DIOs) for the following signals: Transmit Data (TxD) output, Receive Data (RxD) input, Request-To-Send (RTS) output and Clear-To-Send (CTS) input. In 4-wire mode, all four lines are used to implement flow control (this is the default mode). In 2-wire mode, only the TxD and RxD lines are used, and there is no flow control (this mode is not available on the JN5121 device).

The UART functions are listed below, along with their page references:

Function	Page
vAHI_UartEnable	104
vAHI_UartDisable	105
vAHI_UartSetBaudRate	106
vAHI_UartSetBaudDivisor	107
vAHI_UartSetClocksPerBit (JN5148 Only)	108
vAHI_UartSetControl	109
vAHI_UartSetInterrupt	110
vAHI_UartSetRTSCTS (JN5139/JN5148 Only)	111
vAHI_UartSetRTS (JN5148 Only)	112
vAHI_UartSetAutoFlowCtrl (JN5148 Only)	113
vAHI_UartSetBreak (JN5148 Only)	115
vAHI_UartReset	116
u8AHI_UartReadRxFifoLevel (JN5148 Only)	117
u8AHI_UartReadTxFifoLevel (JN5148 Only)	118
u8AHI_UartReadLineStatus	119
u8AHI_UartReadModemStatus	120
u8AHI_UartReadInterruptStatus	121
vAHI_UartWriteData	122
u8AHI_UartReadData	123
vAHI_Uart0RegisterCallback	124
vAHI_Uart1RegisterCallback	125

Note: For guidance on using these functions in JN5148/ JN5139 application code, refer to Chapter 6 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

vAHI_UartEnable

```
void vAHI_UartEnable(uint8 u8Uart);
```

Description

This function enables the specified UART. It must be the first UART function called.

Be sure to enable the UART using this function before writing to the UART using the function **vAHI_UartWriteData()**, otherwise an exception will result.

The UARTs use certain DIO lines, as follows:

UART Signal	DIOs for UART0	DIOs for UART1
CTS	DIO4	DIO17
RTS	DIO5	DIO18
TxD	DIO6	DIO19
RxD	DIO7	DIO20

If a UART uses only the RxD and TxD lines, it is said to operate in 2-wire mode (this mode is not available on the JN5121 device). If, in addition, it uses the RTS and CTS lines to implement flow control, it is said to operate in 4-wire mode.

4-wire mode (with flow control) is enabled by default when **vAHI_UartEnable()** is called. If you wish to implement 2-wire mode on the JN5139 or JN5148 device, you will need to call **vAHI_UartSetRTSCTS()** before calling **vAHI_UartEnable()** in order to release control of the DIOs used for RTS and CTS.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)

Returns

None

vAHI_UartDisable

```
void vAHI_UartDisable(uint8 u8Uart);
```

Description

This function disables the specified UART by powering it down.

Be sure to re-enable the UART using **vAHI_UartEnable()** before attempting to write to the UART using the function **vAHI_UartWriteData()**, otherwise an exception will result.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)

Returns

None

vAHI_UartSetBaudRate

```
void vAHI_UartSetBaudRate(uint8 u8Uart,
                          uint8 u8BaudRate);
```

Description

This function sets the baud-rate for the specified UART to one of a number of standard rates.

The possible baud-rates are:

- 4800 bps
- 9600 bps
- 19200 bps
- 38400 bps
- 76800 bps
- 115200 bps

To set the baud-rate to other values, use the function **vAHI_UartSetBaudDivisor()**.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
u8BaudRate	Desired baud-rate: E_AHI_UART_RATE_4800 (4800 bps) E_AHI_UART_RATE_9600 (9600 bps) E_AHI_UART_RATE_19200 (19200 bps) E_AHI_UART_RATE_38400 (38400 bps) E_AHI_UART_RATE_76800 (76800 bps) E_AHI_UART_RATE_115200 (115200 bps)

Returns

None

vAHI_UartSetBaudDivisor

```
void vAHI_UartSetBaudDivisor(uint8 u8Uart,
                             uint16 u16Divisor);
```

Description

This function sets an integer divisor to derive the baud-rate from a 1-MHz frequency for the specified UART. The function allows baud-rates to be set that are not available through the function **vAHI_UartSetBaudRate()**.

The baud-rate produced is defined by:

baud-rate = 1000000/*u16Divisor*

For example:

u16Divisor	**Baud-rate (bits/s)**
1	1000000
2	500000
9	115200 (approx.)
26	38400 (approx.)

Note that on the JN5148 device, other baud-rates (including higher baud-rates) can be achieved by subsequently calling the function **vAHI_UartSetClocksPerBit()**.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
u16Divisor	Integer divisor

Returns

None

vAHI_UartSetClocksPerBit (JN5148 Only)

void vAHI_UartSetClocksPerBit(uint8 *u8Uart*, **uint8** *u8Cpb***);**

Description

This function sets the baud-rate used by the specified UART on the JN5148 device to a value derived from the 16-MHz clock (assuming sourced from a 32-MHz external crystal oscillator). The function allows higher baud-rates to be set than those available through **vAHI_UartSetBaudRate()** and **vAHI_UartSetBaudDivisor()**.

The obtained baud-rate, in Mbits/s, is given by:

$$\frac{16}{Divisor \times (Cpb + 1)}$$

where *Cpb* is set in this function and *Divisor* is set in **vAHI_UartSetBaudDivisor()**. Therefore, the function **vAHI_UartSetBaudDivisor()** must be called to set *Divisor* before calling **vAHI_UartSetClocksPerBit()**.

Example baud-rates that can be achieved are listed below:

Divisor	*Cpb*	Baud-rate (Mbits/s)
1	3	4.000
1	4	3.200
1	5	2.667
1	6	2.286
1	7	2.000
1	15	1.000
2	11	0.667
2	15	0.500
3	15	0.333

Note that 4 Mbits/s is the highest baud rate that is recommended.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
u8Cpb	*Cpb* value in above formula, in range 0-15 (note that values 0-2 are not recommended)

Returns

None

vAHI_UartSetControl

```
void vAHI_UartSetControl(uint8 u8Uart,
                         bool_t bEvenParity,
                         bool_t bEnableParity,
                         uint8 u8WordLength,
                         bool_t bOneStopBit,
                         bool_t bRtsValue);
```

Description

This function sets various control bits for the specified UART.

Note that RTS cannot be controlled automatically - it can only be set/cleared under software control.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
bEvenParity	Type of parity to be applied (if enabled): E_AHI_UART_EVEN_PARITY (even parity) E_AHI_UART_ODD_PARITY (odd parity)
bEnableParity	Enable/disable parity check: E_AHI_UART_PARITY_ENABLE E_AHI_UART_PARITY_DISABLE
u8WordLength	Word length (in bits): E_AHI_UART_WORD_LEN_5 (word is 5 bits) E_AHI_UART_WORD_LEN_6 (word is 6 bits) E_AHI_UART_WORD_LEN_7 (word is 7 bits) E_AHI_UART_WORD_LEN_8 (word is 8 bits)
bOneStopBit	Number of stop bits - 1 stop bit, or 1.5 or 2 stop bits (depending on word length), enumerated as: E_AHI_UART_1_STOP_BIT (TRUE - 1 stop bit) E_AHI_UART_2_STOP_BITS (FALSE - 1.5 or 2 stop bits)
bRtsValue	Set/clear RTS signal: E_AHI_UART_RTS_HIGH (TRUE - set RTS to high) E_AHI_UART_RTS_LOW (FALSE - clear RTS to low)

Returns

None

vAHI_UartSetInterrupt

```
void vAHI_UartSetInterrupt(uint8 u8Uart,
                           bool_t bEnableModemStatus,
                           bool_t bEnableRxLineStatus,
                           bool_t bEnableTxFifoEmpty,
                           bool_t bEnableRxData,
                           uint8 u8FifoLevel);
```

Description

This function enables or disables the interrupts generated by the specified UART and sets the Receive FIFO trigger-level - that is, the number of bytes required in the Receive FIFO to trigger a 'receive data available' interrupt.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
bEnableModemStatus	Enable/disable 'modem status' interrupt (e.g. CTS change detected): TRUE to enable FALSE to disable
bEnableRxLineStatus	Enable/disable 'receive line status' interrupt (break indication, framing error, parity error or over-run): TRUE to enable FALSE to disable
bEnableTxFifoEmpty	Enable/disable 'Transmit FIFO empty' interrupt: TRUE to enable FALSE to disable
bEnableRxData	Enable/disable 'receive data available' interrupt: TRUE to enable FALSE to disable
u8FifoLevel	Number of bytes in Receive FIFO required to trigger a 'receive data available' interrupt: E_AHI_UART_FIFO_LEVEL_1 (1 byte) E_AHI_UART_FIFO_LEVEL_4 (4 bytes) E_AHI_UART_FIFO_LEVEL_8 (8 bytes) E_AHI_UART_FIFO_LEVEL_14 (14 bytes)

Returns

None

vAHI_UartSetRTSCTS (JN5139/JN5148 Only)

```
void vAHI_UartSetRTSCTS(uint8 u8Uart,
                        bool_t bRTSCTSEn);
```

Description

This function instructs the specified UART to take or release control of the DIO lines used for RTS and CTS in flow control.

UART0:	DIO4 for CTS DIO5 for RTS
UART1:	DIO17 for CTS DIO18 for RTS

The function must be called before **vAHI_UartEnable()** is called.

If a UART uses the RTS and CTS lines, it is said to operate in 4-wire mode, otherwise it is said to operate in 2-wire mode.

On the JN5139 and JN5148 devices, the UARTs operate by default in 4-wire mode. If you wish to use a UART in 2-wire mode, it will be necessary to call **vAHI_UartSetRTSCTS()** before calling **vAHI_UartEnable()** in order to release control of the RTS and CTS lines.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
bRTSCTSEn	Take/release control of DIO lines for RTS and CTS: TRUE to take control FALSE to release control (allow use for other operations)

Returns

None

vAHI_UartSetRTS (JN5148 Only)

void vAHI_UartSetRTS(uint8 *u8Uart*, **bool_t** *bRtsValue***);**

Description

This function instructs the specified UART on the JN5148 device to set or clear its RTS signal.

In order to use this function, the UART must be in 4-wire mode without automatic flow control enabled.

The function must be called after **vAHI_UartEnable()** is called.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
bRtsValue	Set/clear RTS signal: E_AHI_UART_RTS_HIGH (TRUE - set RTS to high) E_AHI_UART_RTS_LOW (FALSE - clear RTS to low)

Returns

None

vAHI_UartSetAutoFlowCtrl (JN5148 Only)

```
void vAHI_UartSetAutoFlowCtrl(uint8 u8Uart,
                              uint8 u8RxFifoLevel,
                              bool_t bFlowCtrlPolarity,
                              bool_t bAutoRts,
                              bool_t bAutoCts);
```

Description

This function allows the Automatic Flow Control (AFC) feature on the JN5148 device to be configured and enabled. The function parameters allow the following to be selected/set:

- **Automatic RTS** (*bAutoRts*): This is the automatic control of the outgoing RTS signal based on the Receive FIFO fill-level. RTS is de-asserted when the Receive FIFO fill-level is greater than or equal to the specified trigger level (*u8RxFifoLevel*). RTS is then re-asserted as soon as Receive FIFO fill-level falls below the trigger level.
- **Automatic CTS** (*bAutoCts*): This is the automatic control of transmissions based on the incoming CTS signal. The transmission of a character is only started if the CTS input is asserted.
- **Receive FIFO Automatic RTS trigger level** (*u8RxFifoLevel*): This is the level at which the outgoing RTS signal is de-asserted when the Automatic RTS feature is enabled (using *bAutoRts*). If using a USB/FTDI cable to connect to the UART, use a setting of 13 bytes or lower (otherwise the Receive FIFO will overflow and data will be lost, as the FTDI device sends up to 3 bytes of data even once RTS has been de-asserted).
- **Flow Control Polarity** (*bFlowCtrlPolarity*): This is the active level (active-low or active-high) of the RTS and CTS hardware flow control signals when using the AFC feature. This setting has no effect when not using AFC (in this case, the software decides the active level, sets the outgoing RTS value and monitors the incoming CTS value).

In order to use the RTS and CTS lines, the UART must be enabled in 4-wire mode, which is the default mode on the JN5148 device.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
u8RxFifoLevel	Receive FIFO automatic RTS trigger level: 00: 8 bytes 01: 11 bytes 10: 13 bytes 11: 15 bytes
bFlowCtrlPolarity	Active level (low or high) of RTS and CTS flow control: FALSE: RTS and CTS are active-low TRUE: RTS and CTS are active-high
bAutoRts	Enable/disable Automatic RTS feature: TRUE to enable FALSE to disable

bAutoCts	Enable/disable Automatic CTS feature: TRUE to enable FALSE to disable

Returns

None

vAHI_UartSetBreak (JN5148 Only)

```
void vAHI_UartSetBreak(uint8 u8Uart, bool_t bBreak);
```

Description

This function instructs the specified UART on the JN5148 device to initiate or clear a transmission break.

On setting the break condition using this function, the data byte that is currently being transmitted is corrupted and transmission then stops. On clearing the break condition, transmission resumes to transfer the data remaining in the Transmit FIFO.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
bBreak	Instruction for UART: TRUE to initiate break (no data) FALSE to clear break (and resume data transmission)

Returns

None

vAHI_UartReset

```
void vAHI_UartReset(uint8 u8Uart,
                    bool_t bTxReset,
                    bool_t bRxReset);
```

Description

This function resets the Transmit and Receive FIFOs of the specified UART. The character currently being transferred is not affected. The Transmit and Receive FIFOs can be reset individually or together.

The function also sets the FIFO trigger-level to single-byte trigger. The Receive FIFO interrupt trigger-level can be set via **vAHI_UartSetInterrupt()**.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
bTxReset	Transmit FIFO reset: TRUE to reset the Transmit FIFO FALSE not to reset the Transmit FIFO
bRxReset	Receive FIFO reset: TRUE to reset the Receive FIFO FALSE not to reset the Receive FIFO

Returns

None

u8AHI_UartReadRxFifoLevel (JN5148 Only)

```
uint8  u8AHI_UartReadRxFifoLevel(uint8 u8Uart);
```

Description

This function obtains the fill-level of the Receive FIFO of the specified UART on the JN5148 device - that is, the number of characters currently in the FIFO.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)

Returns

Number of characters in the specified Receive FIFO

u8AHI_UartReadTxFifoLevel (JN5148 Only)

```
uint8  u8AHI_UartReadTxFifoLevel(uint8 u8Uart);
```

Description

This function obtains the fill-level of the Transmit FIFO of the specified UART on the JN5148 device - that is, the number of characters currently in the FIFO.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)

Returns

Number of characters in the specified Transmit FIFO

u8AHI_UartReadLineStatus

```
uint8 u8AHI_UartReadLineStatus(uint8 u8Uart);
```

Description

This function returns line status information in a bitmap for the specified UART.

Note that the following bits are cleared after reading:

E_AHI_UART_LS_ERROR
E_AHI_UART_LS_BI
E_AHI_UART_LS_FE
E_AHI_UART_LS_PE
E_AHI_UART_LS_OE

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)

Returns

Bitmap:

Bit	Description
E_AHI_UART_LS_ERROR	This bit will be set if a parity error, framing error or break indication has been received
E_AHI_UART_LS_TEMT	This bit will be set if the Transmit Shift Register is empty
E_AHI_UART_LS_THRE	This bit will be set if the Transmit FIFO is empty
E_AHI_UART_LS_BI	This bit will be set if a break indication has been received (line held low for a whole character)
E_AHI_UART_LS_FE	This bit will be set if a framing error has been received
E_AHI_UART_LS_PE	This bit will be set if a parity error has been received
E_AHI_UART_LS_OE	This bit will be set if a receive over-run has occurred, i.e. the receive buffer is full but another character arrives
E_AHI_UART_LS_DR	This bit will be set if there is data in the Receive FIFO

u8AHI_UartReadModemStatus

```
uint8 u8AHI_UartReadModemStatus(uint8 u8Uart);
```

Description

This function obtains modem status information from the specified UART as a bitmap which includes the CTS and ‘CTS has changed’ status (which can be extracted as described below).

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)

Returns

Bitmap in which:

- CTS input status is bit 4 (‘1’ indicates CTS is high, ‘0’ indicates CTS is low).
- ‘CTS has changed’ status is bit 0 (‘1’ indicates that CTS input has changed). If the return value logically ANDed with E_AHI_UART_MS_DCTS is non-zero, the CTS input has changed.

u8AHI_UartReadInterruptStatus

```
uint8 u8AHI_UartReadInterruptStatus(uint8 u8Uart);
```

Description

This function returns a pending interrupt for the specified UART as a bitmap.

Interrupts are returned one at a time, according to their priorities, so there may need to be multiple calls to this function. If interrupts are enabled, the interrupt handler processes this activity and posts each interrupt to the queue or to a callback function.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)

Returns

Bitmap:

Bit range	Value/Enumeration	Description
Bit 0	0	More interrupts pending
	1	No more interrupts pending
Bits 1-3	E_JPI_UART_INT_RXLINE (3)	Receive line status interrupt (highest prioritry)
	E_JPI_UART_INT_RXDATA (2)	Receive data available interrupt (next highest priority)
	E_JPI_UART_INT_TIMEOUT (6)	Timeout interrupt (next highest priority)
	E_JPI_UART_INT_TX (1)	Transmit FIFO empty interrupt (next highest priority)
	E_JPI_UART_INT_MODEM (0)	Modem status interrupt (lowest priority)

The above table lists the UART interrupts (bits 1-3) from highest to lowest priority.

vAHI_UartWriteData

void vAHI_UartWriteData(uint8 *u8Uart*, **uint8** *u8Data***);**

Description

This function writes a data byte to the Transmit FIFO of the specified UART. The data byte will start to be transmitted as soon as it reaches the head of the FIFO.

If no flow control or manual flow control is being implemented for data transmission, the data in the Transmit FIFO will be transmitted as soon as possible (irrespective of the state of the local CTS line). Therefore, the function **vAHI_UartWriteData()** should be called only when the destination device is able to receive the data.

On the JN5148 device, if automatic flow control has been enabled for the local CTS line using the function **vAHI_UartSetAutoFlowCtrl()**, the data in the Transmit FIFO will only be transmitted once the CTS line has been asserted. In this case, **vAHI_UartWriteData()** can be called at any time to load data into the Transmit FIFO, provided that there is enough free space in the FIFO.

Refer to the description of **u8AHI_UartReadTxFifoLevel()** (JN5148 only) or **u8AHI_UartReadLineStatus()** for details of how to determine whether the Transmit FIFO already contains data.

Before this function is called, the UART must be enabled using the function **vAHI_UartEnable()**, otherwise an exception will result.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
u8Data	Byte to transmit

Returns

None

u8AHI_UartReadData

uint8 u8AHI_UartReadData (uint8 *u8Uart***);**

Description

This function returns a single byte read from the Receive FIFO of the specified UART. If the FIFO is empty, the returned value is not valid.

Refer to the description of **u8AHI_UartReadRxFifoLevel()** (JN5148 only) or **u8AHI_UartReadLineStatus()** for details of how to determine whether the Receive FIFO is empty.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)

Returns

Received byte

vAHI_Uart0RegisterCallback

```
void vAHI_Uart0RegisterCallback(
                    PR_HWINT_APPCALLBACK prUart0Callback);
```

Description

This function registers a user-defined callback function that will be called when the UART0 interrupt is triggered.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A.

Parameters

prUart0Callback	Pointer to callback function to be registered

Returns

None

vAHI_Uart1RegisterCallback

```
void vAHI_Uart1RegisterCallback(
                    PR_HWINT_APPCALLBACK prUart1Callback);
```

Description

This function registers a user-defined callback function that will be called when the UART1 interrupt is triggered.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A.

Parameters

prUart1Callback	Pointer to callback function to be registered

Returns

None

7. Timers

This chapter describes the functions that can be used to control the on-chip timers. The number of timers available depends on the device type:

- JN5139 and JN5121 devices have two timers: Timer 0 and Timer 1
- JN5148 has three timers: Timer 0, Timer 1 and Timer 2

They are distinct from the wake timers described in Chapter 8 and tick timer described in Chapter 9.

Modes of Operation

The timers can be operated in the following modes: Timer, Pulse Width Modulation (PWM), Counter, Capture, Delta-Sigma. These modes are summarised in the table below.

Mode	Description
Timer	The source clock is used to produce a pulse cycle defined by the number of clock cycles until a positive pulse edge and until a negative pulse edge. Interrupts can be generated on either or both edges. The pulse cycle can be produced just once in 'single-shot' mode or continuously in 'repeat' mode.
PWM	As for Timer mode, except the Pulse Width Modulated signal is output on a DIO (which depends on the specific timer used - see DIO Usage below).
Counter	The timer is used to count edges on an external input signal, selected as an external clock input. The timer can count just positive edges or both positive and negative edges.
Capture	An external input signal is sampled on every tick of the source clock. The results of the capture allow the period and pulse width of the sampled signal to be calculated. If required, the results can be read without stopping the timer.
Delta-Sigma	The timer is used as a low-rate DAC. The converted signal is output on a DIO (which depends on the specific timer used - see DIO Usage below) and requires simple filtering to give the analogue signal.

Table 1: Modes of Timer Operation

The timers normally use the internal 16-MHz clock source, but can alternatively use an external clock source (which should only be used for Counter mode).

Caution: *You must enable a timer before attempting any other operation on it, otherwise an exception may result.*

DIO Usage

The timers use certain DIO pins, as indicated in the table below.

Timer 0 DIO	Timer 1 DIO	Timer 2 DIO (JN5148 Only)	Function
8	11*	Not Applicable**	Clock or gate input
9	12	Not Applicable**	Capture input
10	13	11*	PWM output

Table 2: DIO Usage with Timers

* DIO11 is shared by Timer 1 and TImer 2 on the JN5148 device, and their use must not conflict.

** Timer 2 (JN5148 only) has no inputs.

The availability of the above DIOs for general-purpose IO when the timers are in use is summarised in the table below for the different Jennic devices.

Device	DIO Availability
JN5121	When enabled, the timer always uses the assigned DIOs. Therefore, none of these DIOs will be available for general-purpose IO.
JN5139	When enabled, the timer uses all or none of the assigned DIOs, as configured using the function **vAHI_TimerDIOControl()**. Therefore, none or all of these DIOs will be available for general-purpose IO.
JN5148	When enabled, the timer can use individual DIOs selected using the function **vAHI_TimerFineGrainDIOControl()** - the remaining DIOs can be used for general-purpose IO. Alternatively, the timer can use all or none of the assigned DIOs, as configured using the function **vAHI_TimerDIOControl()**.

Table 3: DIO Availability During Timer Use

The timer functions are listed below, along with their page references:

Function	Page
vAHI_TimerEnable	130
vAHI_TimerClockSelect (JN5148 Only)	132
vAHI_TimerConfigureOutputs (JN5148 Only)	133
vAHI_TimerConfigureInputs (JN5148 Only)	134
vAHI_TimerStartSingleShot	135
vAHI_TimerStartRepeat	136
u16AHI_TimerReadCount	140
vAHI_TimerStartDeltaSigma	138
u16AHI_TimerReadCount	140
vAHI_TimerReadCapture	141
vAHI_TimerReadCaptureFreeRunning	142
vAHI_TimerStop	143
vAHI_TimerDisable	144
vAHI_TimerDIOControl (JN5139/JN5148 Only)	145
vAHI_TimerFineGrainDIOControl (JN5148 Only)	146
u8AHI_TimerFired	147
vAHI_Timer0RegisterCallback	148
vAHI_Timer1RegisterCallback	149
vAHI_Timer2RegisterCallback (JN5148 Only)	150

Note: For guidance on using these functions in JN5148/ JN5139 application code, refer to Chapter 7 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

vAHI_TimerEnable

```
void vAHI_TimerEnable(uint8 u8Timer,
                      uint8 u8Prescale,
                      bool_t bIntRiseEnable,
                      bool_t bIntPeriodEnable,
                      bool_t bOutputEnable);
```

Description

This function configures and enables the specified timer, and must be the first timer function called. The timer is derived from the 16-MHz system clock, which can be divided down to produce the timer clock. The timer can be used in various modes, described in Table 1 on page 127.

The parameters of this enable function cover the following features:

- **Prescaling** (*u8Prescale*): The timer's source clock is divided down to produce a slower clock for the timer, the divisor being $2^{Prescale}$. Therefore:

  Timer clock frequency = Source clock frequency / $2^{Prescale}$
- **Interrupts** (*bIntRiseEnable* and *bIntPeriodEnable*): Interrupts can be generated:
  - in Timer or PWM mode, on a low-to-high transition (rising output) and/or on a high-to-low transition (end of the timer period)
  - in Counter mode, on reaching target counts

  You can register a user-defined callback function for timer interrupts using the function **vAHI_Timer0RegisterCallback()** for Timer 0, **vAHI_Timer1RegisterCallback()** for Timer 1 or **vAHI_Timer2RegisterCallback()** for Timer 2. Alternatively, timer interrupts can be disabled.
- **Timer output** (*bOutputEnable*): When operating in PWM mode or Delta-Sigma mode, the timer's signal is output on a DIO pin (DIO10 for Timer 0, DIO13 for Timer 1, DIO11 for Timer 2), which must be enabled. If this option is enabled, the other DIOs associated with the timer cannot be used for general-purpose input/output.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (Timer 2 - JN5148 only)
u8Prescale	Prescale index, in range 0 to 16, used in dividing down source clock (divisor is $2^{Prescale}$)
bIntRiseEnable	Enable/disable interrupt on rising output (low-to-high): TRUE to enable FALSE to disable
bIntPeriodEnable	Enable/disable interrupt at end of timer period (high-to-low): TRUE to enable FALSE to disable
bOutputEnable	Enable/disable output of timer signal on DIO: TRUE to enable (PWM or Delta-Sigma mode) FALSE to disable (Timer mode)

Returns

None

vAHI_TimerClockSelect (JN5148 Only)

```
void vAHI_TimerClockSelect(uint8 u8Timer,
                           bool_t bExternalClock,
                           bool_t bInvertClock);
```

Description

This function can be used to enable/disable an external clock input for Timer 0 or Timer 1 on the JN5148 device. If enabled, the external input in taken from DIO8 for Timer 0 or from DIO11 for Timer 1 (Timer 2 cannot take an external clock input).

Note the following:

- This function should only be called when using the timer in Counter mode - in this mode, the timer is used to count edges on an input clock or pulse train.
- Output gating can be enabled when the internal clock is used.

If required, this function must be called after **vAHI_TimerEnable()**.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (Timer 2 - JN5148 only)
bExternalClock	Clock source: TRUE to use an external source (Counter mode only) FALSE to use the internal 16-MHz clock
bInvertClock	TRUE to gate the output pin when the gate input is high and invert the clock FALSE to gate the output pin when the gate input is low and not invert the clock

Returns

None

vAHI_TimerConfigureOutputs (JN5148 Only)

void vAHI_TimerConfigureOutputs(uint8 *u8Timer*,
bool_t *bInvertPwmOutput*,
bool_t *bGateDisable***);**

Description

This function configures certain parameters relating to the operation of the specified timer on the JN5148 device in the following modes (described in Table 1 on page 127):

- Timer mode, in which the internal system clock drives the timer's counter in order to produce a pulse cycle in either 'single shot' or 'repeat' mode. The clock may be temporarily interrupted by a gating input on DIO8 for Timer 0 or DIO11 for Timer 1 (there is no gating input for Timer 2). Clock gating is enabled/disabled using this function.
- Pulse Width Modulation (PWM) mode, in which the PWM signal produced in Timer mode (see above) is output - this output can be enabled in **vAHI_TimerEnable()**. The signal is output on a DIO which depends on the timer selected - DIO10 for Timer 0, DIO13 for Timer 1 and DIO11 for Timer 2. If required, the output signal can be inverted using this function.

This function must be called after the specified timer has been enabled through **vAHI_TimerEnable()** and before the timer is started.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (Timer 2)
bInvertPwmOutput	Enable/disable inversion of PWM output: TRUE to enable inversion FALSE to disable inversion
bGateDisable	Enable/disable external gating input for Timer mode: TRUE to disable clock gating input FALSE to enable clock gating input

Returns

None

vAHI_TimerConfigureInputs (JN5148 Only)

```
void vAHI_TimerConfigureInputs(uint8 u8Timer,
                               bool_t bInvCapt,
                               bool_t bEventEdge);
```

Description

This function configures certain parameters relating to the operation of the specified timer on the JN5148 device in the following modes (described in Table 1 on page 127):

- Capture mode, in which an external signal is sampled on every tick of the timer. The results of the capture allow the period and pulse width of the sampled signal to be obtained. The input signal can be inverted using this function, allowing the low-pulse width to be measured (instead of the high-pulse width). This external signal is input on a DIO which depends on the timer selected - DIO9 for Timer 0 and DIO12 for Timer 1 (Timer 2 on the JN5148 device cannot be used for capture mode).
- Counter mode, in which the timer is used to count the number of transitions on an external input (selected using **vAHI_TimerClockSelect()**). This configure function allows selection of the transitions on which the count will be performed - on low-to-high transitions, or on both low-to-high and high-to-low transitions.

This function must be called after the specified timer has been enabled through **vAHI_TimerEnable()** and before the timer is started.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1)
bInvCapt	Enable/disable inversion of the capture input signal: TRUE to enable inversion FALSE to disable inversion
bEventEdge	Determines the edge(s) of the external input on which the count will be incremented in counter mode: TRUE - on both low-to-high and high-to-low transitions FALSE - on low-to-high transition

Returns

None

vAHI_TimerStartSingleShot

```
void vAHI_TimerStartSingleShot(uint8 u8Timer,
                               uint16 u16Hi,
                               uint16 u16Lo);
```

Description

This function starts the specified timer in 'single-shot' mode. The function relates to Timer mode, PWM mode and Counter mode, described in Table 1 on page 127.

In **Timer** or **PWM mode**, during one pulse cycle produced, the timer signal starts low and then goes high:

1. The output is low until *u16Hi* clock periods have passed, when it goes high.
2. The output remains high until *u16Lo* clock periods have passed since the timer was started and then goes low again (marking the end of the pulse cycle).

If enabled through **vAHI_TimerEnable()**, an interrupt can be triggered at the low-high transition and/or the high-low transition.

In **Counter mode** (Timer 0 and Timer 1 only), this function is used differently:

- At a count of *u16Hi*, an interrupt (E_AHI_TIMER_RISE_MASK) will be generated (if enabled).
- At a count of *u16Lo*, another interrupt (E_AHI_TIMER_PERIOD_MASK) will be generated (if enabled) and the timer will stop.

Again, interrupts are enabled through **vAHI_TimerEnable()**.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (Timer 2 - JN5148 only)
u16Hi	Number of clock periods after starting a timer before the output goes high (Timer or PWM mode) or count at which first interrupt generated (Counter mode)
u16Lo	Number of clock periods after starting a timer before the output goes low again (Timer or PWM mode) or count at which second interrupt generated and timer stops (Counter mode)

Returns

None

vAHI_TimerStartRepeat

```
void vAHI_TimerStartRepeat(uint8 u8Timer,
                           uint16 u16Hi,
                           uint16 u16Lo);
```

Description

This function starts the specified timer in ‘repeat’ mode. The function relates to Timer mode, PWM mode and Counter mode, described in Table 1 on page 127.

In **Timer** or **PWM mode**, during each pulse cycle produced, the timer signal starts low and then goes high:

1. The output is low until *u16Hi* clock periods have passed, when it goes high.
2. The output remains high until *u16Lo* clock periods have passed since the timer was started and then goes low again.

The above process repeats until the timer is stopped using **vAHI_TimerStop()**.

If enabled through **vAHI_TimerEnable()**, an interrupt can be triggered at the low-high transition and/or the high-low transition.

In **Counter mode** (Timer 0 and Timer 1 only), this function is used differently:

- At a count of *u16Hi*, an interrupt (E_AHI_TIMER_RISE_MASK) will be generated (if enabled).
- At a count of *u16Lo*, another interrupt (E_AHI_TIMER_PERIOD_MASK) will be generated (if enabled) and the count will then be re-started from zero.

Again, interrupts are enabled through **vAHI_TimerEnable()**.

The current count can be read at any time using **u16AHI_TimerReadCount**.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (Timer 2 - JN5148 only)
u16Hi	Number of clock periods after starting a timer before the output goes high (Timer or PWM mode) or count at which first interrupt generated (Counter mode)
u16Lo	Number of clock periods after starting a timer before the output goes low again (Timer or PWM mode) or count at which second interrupt generated (Counter mode)

Returns

None

vAHI_TimerStartCapture

```
void vAHI_TimerStartCapture(uint8 u8Timer);
```

Description

This function starts the specified timer in Capture mode. This mode must first be configured using the function **vAHI_TimerConfigureInputs()**.

An input signal must be provided on pin DIO9 for Timer 0 or DIO12 for Timer 1 (Capture mode is not available on Timer 2 of the JN5148 device). The incoming signal is timed and the captured measurements are:

- number of clock cycles to the last low-to-high transition of the input signal
- number of clock cycles to the last high-to-low transition of the input signal

These values are placed in registers to be read later using the function **vAHI_TimerReadCapture()** or **vAHI_TimerReadCaptureFreeRunning()**. They allow the input pulse width to be determined.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1)

Returns

None

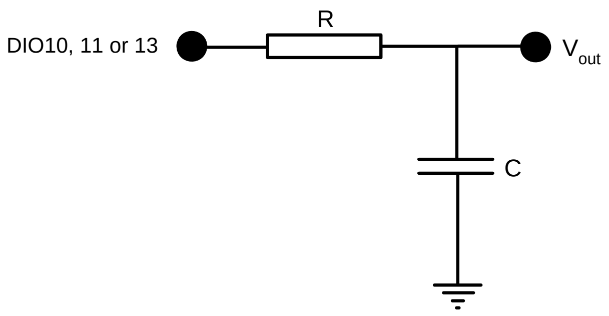
vAHI_TimerStartDeltaSigma

void vAHI_TimerStartDeltaSigma(uint8 *u8Timer*,
uint16 *u16Hi*,
uint16 *0x0000*,
bool_t *bRtzEnable*);

Description

This function starts the specified timer in Delta-Sigma mode, which allows the timer to be used as a low-rate DAC.

To use this mode, the relevant DIO output for the timer (DIO10 for Timer 0, DIO13 for Timer 1, DIO11 for Timer 2) must be enabled through **vAHI_TimerEnable()**. In addition, an RC circuit must be inserted on the DIO output pin in the arrangement shown below (also see Note below).



The 16-MHz system clock is used as the timer source and the conversion period of the ‘DAC’ is 65536 clock cycles. In Delta-Sigma mode, the timer outputs a number of randomly spaced clock pulses as specified by the value being converted. When RC-filtered, this produces an analogue voltage proportional to the conversion value.

If the RTZ (Return-to-Zero) option is enabled, a low clock cycle is inserted after every clock cycle, so that there are never two consecutive high clock cycles. This doubles the conversion period, but improves linearity if the rise and fall times of the outputs are different from one another.

Note: For more information on ‘Delta-Sigma’ mode, refer to the data sheet for your Jennic wireless microcontroller. Also, refer to the Application Note *Using JN51xx Timers (JN-AN-1032)*, which includes the selection of the above R and C values.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (Timer 2 - JN5148 only)
u16Hi	Number of 16-MHz clock cycles for which the output will be high during a conversion period, in the range 0 to 65535 (full period is 65536 clock cycles)
0x0000	Fixed null value
bRtzEnable	Enable/disable RTZ (Return-to-Zero) option: TRUE to enable FALSE to disable

Returns

None

u16AHI_TimerReadCount

uint16 u16AHI_TimerReadCount(uint8 *u8Timer***);**

Description

This function obtains the current count value of the specified timer.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (Timer 2 - JN5148 only)

Returns

Current count value of timer

vAHI_TimerReadCapture

```
void vAHI_TimerReadCapture(uint8 u8Timer,
                           uint16 *pu16Hi,
                           uint16 *pu16Lo);
```

Description

This function stops the specified timer and then obtains the results from a 'capture' started using the function **vAHI_TimerStartCapture()**.

The values returned are offsets from when the capture was originally started, as follows:

- number of clock cycles to the last low-to-high transition of the input signal
- number of clock cycles to the last high-to-low transition of the input signal

The width of the last pulse can be calculated from the difference of these results, provided that the results were requested during a low period. However, since it is not possible to be sure of this, the results obtained from this function may not always be valid for calculating the pulse width.

If you wish to measure the pulse period of the input signal, you should use the function **vAHI_TimerReadCaptureFreeRunning()**, which does not stop the timer.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1)
**pu16Hi*	Pointer to location which will receive clock period at which last low-high transition occurred
**pu16Lo*	Pointer to location which will receive clock period at which last high-low transition occurred

Returns

None

vAHI_TimerReadCaptureFreeRunning

```
void vAHI_TimerReadCaptureFreeRunning(uint8 u8Timer,
                                      uint16 *pu16Hi,
                                      uint16 *pu16Lo);
```

Description

This function obtains the results from a 'capture' started using the function **vAHI_TimerStartCapture()**. This function does not stop the timer.

Alternatively, the function **vAHI_TimerReadCapture()** can be used, which stops the timer before reporting the capture measurements.

The values returned are offsets from when the capture was originally started, as follows:

- number of clock cycles to the last low-to-high transition of the input signal
- number of clock cycles to the last high-to-low transition of the input signal

The width of the last pulse can be calculated from the difference of these results, provided that the results were requested during a low period. However, since it is not possible to be sure of this, the results obtained from this function may not always be valid for calculating the pulse width.

If you wish to measure the pulse period of the input signal, you should call this function twice during consecutive pulse cycles. For example, a call to this function could be triggered by an interrupt generated on a particular type of transition (low-to-high or high-to-low). The pulse period can then be obtained by calculating the difference between the results for consecutive low-to-high transitions or the difference between the results for consecutive high-to-low transitions.

***Caution:** Since it is not possible to be sure of the state of the input signal when capture started, the results of the first call to this function after starting capture should be discarded.*

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1)
**pu16Hi*	Pointer to location which will receive clock period at which last low-high transition occurred
**pu16Lo*	Pointer to location which will receive clock period at which last high-low transition occurred

Returns

None

vAHI_TimerStop

```
void vAHI_TimerStop (uint8 u8Timer);
```

Description

This function stops the specified timer.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (Timer 2 - JN5148 only)

Returns

None

vAHI_TimerDisable

void vAHI_TimerDisable (uint8 *u8Timer***);**

Description

This function disables the specified timer. As well as stopping the timer from running, the clock to the timer block is switched off in order to reduce power consumption. This means that any subsequent attempt to access the timer will be unsuccessful until **vAHI_TimerEnable()** is called to re-enable the block.

Caution: *An attempt to access the timer while it is disabled will result in an exception.*

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (Timer 2 - JN5148 only)

Returns

None

vAHI_TimerDIOControl (JN5139/JN5148 Only)

```
void vAHI_TimerDIOControl(uint8 u8Timer,
                          bool_t bDIOEnable);
```

Description

This function enables/disables DIOs for use by the specified timer (Timer 0 or 1) on the JN5139 and JN5148 devices:

- DIO8, DIO9 and DIO10 for Timer 0
- DIO11, DIO12 and DIO13 for Timer 1

Refer to the table at the start of this chapter for the timer signals on these DIOs.

The function configures the set of three DIOs for a timer. By default, all these DIOs are enabled for timer use. If disabled, the DIOs can be used as GPIOs (General Purpose Inputs/Outputs). You should perform this configuration before the timers are enabled using **vAHI_TimerEnable()**, in order to avoid glitching on the GPIOs during timer operation.

On the JN5148 device, you can use the function **AHI_TimerFineGrainDIOControl()** to configure the use of all the DIOs for all the timers in one call, including Timer 2, and can individually enable/disable the DIOs.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1)
bDIOEnable	Enable/disable use of associated DIOs by timer: TRUE to enable FALSE to disable (so available as GPIOs)

Returns

None

vAHI_TimerFineGrainDIOControl (JN5148 Only)

void vAHI_TimerFineGrainDIOControl(uint8 *u8BitMask***);**

Description

This function allows the DIOs associated with the timers on the JN5148 device to be enabled/disabled for use by the timers. The function allows the DIOs for all the timers to be configured in one call: Timer 0, Timer 1 and Timer 2.

By default, all these DIOs are enabled for timer use. Therefore, you can use this function to release those DIOs that you do not wish to use for the timers. The released DIOs will then be available as GPIOs (General Purpose Inputs/Outputs). You should perform this configuration before the timers are enabled using **vAHI_TimerEnable()**, in order to avoid glitching on the GPIOs during timer operation.

The DIO configuration information is passed into the function as an 8-bit bitmap. The individual bit assigments are detailed in the table below. A bit is set to 1 to disable the corresponding DIO and is set to 0 to enable the DIO for timer use.

Bit	Timer Input/Output and DIO
0	Timer 0 external gate/event input on DIO8
1	Timer 0 capture input on DIO9
2	Timer 0 PWM output on DIO10
3	Timer 1 external gate/event input on DIO11
4	Timer 1 capture input on DIO12
5	Timer 1 PWM output on DIO13
6	Timer 2 PWM output on DIO11
7	Reserved

Note: DIO11 is shared between Timer 1 and Timer 2. If this DIO is enabled for use by both timers, Timer 2 will take precedence.

Parameters

u8BitMask	Bitmap containing DIO configuration information for all timers (see above)

Returns

None

u8AHI_TimerFired

```
uint8 u8AHI_TimerFired(uint8 u8Timer);
```

Description

This function obtains the interrupt status of the specified timer. The function also clears interrupt status after reading it.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (Timer 2 - JN5148 only)

Returns

Bitmap:

Returned value logical ANDed with E_AHI_TIMER_RISE_MASK - will be non-zero if interrupt for low-to-high transition (output rising) has been set

Returned value logical ANDed with E_AHI_TIMER_PERIOD_MASK - will be non-zero if interrupt for high-to-low transition (end of period) has been set

vAHI_Timer0RegisterCallback

```
void vAHI_Timer0RegisterCallback(
            PR_HWINT_APPCALLBACK PrTimer0Callback);
```

Description

This function registers a user-defined callback function that will be called when the Timer 0 interrupt is triggered.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A.

Parameters

PrTimer0Callback	Pointer to callback function to be registered

Returns

None

vAHI_Timer1RegisterCallback

```
void vAHI_Timer1RegisterCallback(
            PR_HWINT_APPCALLBACK PrTimer1Callback);
```

Description

This function registers a user-defined callback function that will be called when the Timer 1 interrupt is triggered.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A.

Parameters

PrTimer1Callback	Pointer to callback function to be registered

Returns

None

vAHI_Timer2RegisterCallback (JN5148 Only)

```
void vAHI_Timer2RegisterCallback(
                PR_HWINT_APPCALLBACK PrTimer2Callback);
```

Description

This function registers a user-defined callback function that will be called when the Timer 2 interrupt is triggered on the JN5148 device.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A.

Parameters

PrTimer2Callback	Pointer to callback function to be registered

Returns

None

8. Wake Timers

This chapter details the functions for controlling the wake timers. The Jennic wireless microcontrollers include two wake timers, denoted Wake Timer 0 and Wake Timer 1. These are 35-bit timers on the JN5148 device, and 32-bit timers on the JN5139 and JN5121 devices.

The wake timers are normally used on End Devices to time sleep periods and can be programmed to generate interrupts when the timeout period is reached. They can also be used outside of sleep periods, while the CPU is running (although there is another set of timers with more functionality that can operate only while the CPU is running - see Chapter 7).

The wake timers run at a nominal 32 kHz. On the JN5148 device, their 32-kHz clock source is selectable using the function **bAHI_Set32KhzClockMode()** described on page 42 (this clock selection is preserved during sleep). The wake timers may run up to 30% fast or slow depending on temperature, supply voltage and manufacturing tolerance. For situations in which accurate timing is required, a self-calibration facility is provided to time the 32-kHz clock against the 16-MHz system clock.

The wake timer functions are listed below, along with their page references:

Function	**Page**
vAHI_WakeTimerEnable	152
vAHI_WakeTimerStart (JN5139/JN5121 Only)	153
vAHI_WakeTimerStartLarge (JN5148 Only)	154
vAHI_WakeTimerStop	155
u32AHI_WakeTimerRead (JN5139/JN5121 Only)	156
u64AHI_WakeTimerReadLarge (JN5148 Only)	157
u8AHI_WakeTimerStatus	158
u8AHI_WakeTimerFiredStatus	159
u32AHI_WakeTimerCalibrate	160

Note: For guidance on using these functions in JN5148/ JN5139 application code, refer to Chapter 8 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

vAHI_WakeTimerEnable

```
void vAHI_WakeTimerEnable(uint8 u8Timer,
                          bool_t bIntEnable);
```

Description

This function allows the wake timer interrupt (which is generated when the timer fires) to be enabled/disabled. If this function is called for a wake timer that is already running, it will stop the wake timer.

The wake timer can be subsequently started using the function **vAHI_WakeTimerStart()**.

Wake timer interrupts are handled by the System Controller callback function, registered using the function **vAHI_SysCtrlRegisterCallback()**.

Parameters

u8Timer	Identity of timer: E_AHI_WAKE_TIMER_0 (Wake Timer 0) E_AHI_WAKE_TIMER_1 (Wake Timer 1)
bIntEnable	Interrupt enable/disable: TRUE to enable interrupt when wake timer fires FALSE to disable interrupt

Returns

None

vAHI_WakeTimerStart (JN5139/JN5121 Only)

void vAHI_WakeTimerStart(uint8 *u8Timer***,**
uint32 *u32Count***);**

Description

This function starts the specified 32-bit wake timer with the specified count value on the JN5139/JN5121 device. The wake timer will count down from this value, which is set according to the desired timer duration. On reaching zero, the timer ‘fires’, rolls over to 0xFFFFFFFF and continues to count down.

The count value, *u32Count*, is set as the required number of 32-kHz periods. Thus:

Timer duration (in seconds) = *u32Count* / 32000

Note that the 32-kHz internal clock, which drives the wake timer, may be running up to 30% fast or slow. For accurate timings, you are advised to first calibrate the clock using the function **u32AHI_WakeTimerCalibrate()** and adjust the specified count value accordingly.

If you wish to enable interrupts for the wake timer, you must call **vAHI_WakeTimerEnable()** before calling **vAHI_WakeTimerStart()**. The wake timer can be subsequently stopped using **vAHI_WakeTimerStop()** and can be read using **u32AHI_WakeTimerRead()**. Stopping the timer does not affect interrupts that have been set using **vAHI_WakeTimerEnable()**.

Parameters

u8Timer	Identity of timer: E_AHI_WAKE_TIMER_0 (Wake Timer 0) E_AHI_WAKE_TIMER_1 (Wake Timer 1)
u32Count	Count value in 32-kHz periods, i.e. 32 is 1 millisecond (values of 0 and 1 must not be used)

Returns

None

vAHI_WakeTimerStartLarge (JN5148 Only)

void vAHI_WakeTimerStartLarge(uint8 *u8Timer***,**
uint64 *u64Count***);**

Description

This function starts the specified 35-bit wake timer with the specified count value on the JN5148 device. The wake timer will count down from this value, which is set according to the desired timer duration. On reaching zero, the timer 'fires', rolls over to 0x7FFFFFFFF and continues to count down.

The count value, *u64Count*, is set as the required number of 32-kHz periods. Thus:

Timer duration (in seconds) = *u64Count* / 32000

Note that the 32-kHz internal clock, which drives the wake timer, may be running up to 30% fast or slow. For accurate timings, you are advised to first calibrate the clock using the function **u32AHI_WakeTimerCalibrate()** and adjust the specified count value accordingly.

If you wish to enable interrupts for the wake timer, you must call **vAHI_WakeTimerEnable()** before calling **vAHI_WakeTimerStartLarge()**. The wake timer can be subsequently stopped using **vAHI_WakeTimerStop()** and can be read using **u64AHI_WakeTimerReadLarge()**. Stopping the timer does not affect interrupts that have been set using **vAHI_WakeTimerEnable()**.

Parameters

u8Timer	Identity of timer: E_AHI_WAKE_TIMER_0 (Wake Timer 0) E_AHI_WAKE_TIMER_1 (Wake Timer 1)
u64Count	Count value in 32-kHz periods, i.e. 32 is 1 millisecond. This value must not exceed 0x7FFFFFFFF, and the values 0 and 1 must not be used

Returns

None

vAHI_WakeTimerStop

```
void vAHI_WakeTimerStop(uint8 u8Timer);
```

Description

This function stops the specified wake timer.

Note that no interrupt will be generated.

Parameters

u8Timer	Identity of timer: E_AHI_WAKE_TIMER_0 (Wake Timer 0) E_AHI_WAKE_TIMER_1 (Wake Timer 1)

Returns

None

u32AHI_WakeTimerRead (JN5139/JN5121 Only)

```
uint32 u32AHI_WakeTimerRead(uint8 u8Timer);
```

Description

This function obtains the current value of the specified 32-bit wake timer counter (which counts down) on the JN5139/JN5121 device, without stopping the counter.

Note that on reaching zero, the timer ‘fires’, rolls over to 0xFFFFFFFF and continues to count down. The count value obtained using this function then allows the application to calculate the time that has elapsed since the wake timer fired.

Parameters

u8Timer	Identity of timer: E_AHI_WAKE_TIMER_0 (Wake Timer 0) E_AHI_WAKE_TIMER_1 (Wake Timer 1)

Returns

Current value of wake timer counter

u64AHI_WakeTimerReadLarge (JN5148 Only)

```
uint64 u64AHI_WakeTimerReadLarge(uint8 u8Timer);
```

Description

This function obtains the current value of the specified 35-bit wake timer counter (which counts down) on the JN5148 device, without stopping the counter.

Note that on reaching zero, the timer ‘fires’, rolls over to 0x7FFFFFFFF and continues to count down. The count value obtained using this function then allows the application to calculate the time that has elapsed since the wake timer fired.

Parameters

u8Timer	Identity of timer: E_AHI_WAKE_TIMER_0 (Wake Timer 0) E_AHI_WAKE_TIMER_1 (Wake Timer 1)

Returns

Current value of wake timer counter

u8AHI_WakeTimerStatus

```
uint8 u8AHI_WakeTimerStatus(void);
```

Description

This function determines which wake timers are active. It is possible to have more than one wake timer active at the same time. The function returns a bitmap where the relevant bits are set to show which wake timers are active.

Note that a wake timer remains active after its countdown has reached zero (when the timer rolls over to 0xFFFFFFFF and continues to count down).

Parameters

None

Returns

Bitmap:

Returned value logical ANDed with E_AHI_WAKE_TIMER_MASK_0 will be non-zero if Wake Timer 0 is active

Returned value logical ANDed with E_AHI_WAKE_TIMER_MASK_1 will be non-zero if Wake Timer 1 is active

u8AHI_WakeTimerFiredStatus

```
uint8 u8AHI_WakeTimerFiredStatus(void);
```

Description

This function determines which wake timers have fired (by having passed zero). The function returns a bitmap where the relevant bits are set to show which timers have fired. Any fired timer status is cleared as a result of this call.

Note: If you wish to use this function to check whether a wake timer caused a wake-up event, you must call it before **u32AHI_Init()**. Alternatively, you can determine the wake source as part of your System Controller callback function. For more information, refer to Appendix A.

Parameters

None

Returns

Bitmap:

Returned value logical ANDed with E_AHI_WAKE_TIMER_MASK_0 will be non-zero if Wake Timer 0 has fired
Returned value logical ANDed with E_AHI_WAKE_TIMER_MASK_1 will be non-zero if Wake Timer 1 has fired

u32AHI_WakeTimerCalibrate

```
uint32 u32AHI_WakeTimerCalibrate(void);
```

Description

This function requests a calibration of the 32-kHz internal clock (on which the wake timers run) against the more accurate 16-MHz system clock. Note that the 32-kHz clock has a tolerance of ±30%.

This function uses Wake Timer 0 and takes twenty 32-kHz clock periods to complete the calibration.

The returned result, n, is interpreted as follows:

- n = 10000 ⇒ clock running at 32 kHz
- n > 10000 ⇒ clock running slower than 32 kHz
- n < 10000 ⇒ clock running faster than 32 kHz

The returned value can be used to adjust the time interval value used to program a wake timer. If the required timer duration is T seconds, the count value N that must be specified in **vAHI_WakeTimerStart()** or **vAHI_WakeTimerStartLarge()** is given by N = (10000/n) x 32000 x T.

Parameters

None

Returns

Calibration measurement, n (see above)

9. Tick Timer

This chapter details the functions for controlling the Tick Timer on the Jennic wireless microcontrollers - this is a hardware timer, derived from the 16-MHz system clock. It can be used to generate timing interrupts to software.

The Tick Timer can be used to implement:

- regular events, such as ticks for software timers or an operating system
- a high-precision timing reference
- system monitor timeouts, as used in a watchdog timer

Note: On the JN5139 and JN5121 devices, the Tick Timer cannot be used to bring the CPU out of doze mode.

The tick timer functions are listed below, along with their page references:

Function	Page
vAHI_TickTimerConfigure	162
vAHI_TickTimerInterval	163
vAHI_TickTimerWrite	164
u32AHI_TickTimerRead	165
vAHI_TickTimerIntEnable	166
bAHI_TickTimerIntStatus	167
vAHI_TickTimerIntPendClr	168
vAHI_TickTimerInit (JN5139/JN5121 Only)	169
vAHI_TickTimerRegisterCallback (JN5148 Only)	170

Note: For guidance on using these functions in JN5148/ JN5139 application code, refer to Chapter 9 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

vAHI_TickTimerConfigure

void vAHI_TickTimerConfigure(uint8 *u8Mode***);**

Description

This function configures the operating mode of the Tick Timer and enables the timer. It can also be used to disable the timer.

The Tick Timer counts upwards until the count matches a pre-defined reference value. This function determines what the timer will do once the reference count has been reached. The options are:

- Continue counting upwards
- Restart the count from zero
- Stop counting (single-shot mode)

The reference count is set using the function **vAHI_TickTimerInterval()**. An interrupt can be enabled which is generated on reaching the reference count - see the description of **vAHI_TickTimerIntEnable()**.

The Tick Timer will start running as soon as **vAHI_TickTimerConfigure()** enables it in one of the above modes, irrespective of the state of its counter. In practice, to use the Tick Timer:

1. Call **vAHI_TickTimerConfigure()** to disable the Tick Timer.
2. Call **vAHI_TickTimerWrite()** to set an appropriate starting value for the count.
3. Call **vAHI_TickTimerInterval()** to set the reference count.
4. Call **vAHI_TickTimerConfigure()** again to start the Tick Timer in the desired mode.

On device power-up/reset, the Tick Timer is disabled. However, you are advised to always follow the above sequence of function calls to start the timer.

If the Tick Timer is enabled in single-shot mode, once it has stopped (on reaching the reference count), it can be started again simply by setting another starting value using **vAHI_TickTimerWrite()**.

Parameters

u8Mode	Tick Timer operating mode Action to take on reaching reference count: E_AHI_TICK_TIMER_CONT (continue counting) E_AHI_TICK_TIMER_RESTART (restart from zero) E_AHI_TICK_TIMER_STOP (stop timer) Disable timer: E_AHI_TICK_TIMER_DISABLE (disable timer)

Returns

None

vAHI_TickTimerInterval

```
void vAHI_TickTimerInterval(uint32 u32Interval);
```

Description

This function sets the 28-bit reference count for the Tick Timer.

This is the value with which the actual count of the Tick Timer is compared. The action taken when the count reaches this reference value is determined using the function **vAHI_TickTimerConfigure()**. An interrupt can be also enabled which is generated on reaching the reference count - see the function **vAHI_TickTimerIntEnable()**.

Parameters

u32Interval	Tick Timer reference count (in the range 0 to 0x0FFFFFFF)

Returns

None

vAHI_TickTimerWrite

```
void vAHI_TickTimerWrite(uint32 u32Count);
```

Description

This function sets the initial count of the Tick Timer. If the timer is enabled, it will immediately start counting from this value.

By specifying a count of zero, the function can be used to reset the Tick Timer count to zero at any time.

Parameters

u32Count	Tick Timer count (in the range 0 to 0xFFFFFFFF)

Returns

None

u32AHI_TickTimerRead

```
uint32 u32AHI_TickTimerRead(void);
```

Description

This function obtains the current value of the Tick Timer counter.

Parameters

None

Returns

Value of the Tick Timer counter

vAHI_TickTimerIntEnable

```
void vAHI_TickTimerIntEnable(bool_t bIntEnable);
```

Description

This function can be used to enable Tick Timer interrupts, which are generated when the Tick Timer count reaches the reference count specified using the function **vAHI_TickTimerInterval()**.

A user-defined callback function, which is invoked when the interrupt is generated, can be registered using the function **vAHI_TickTimerRegisterCallback()** for JN5148 or **vAHI_TickTimerInit()** for JN5139 and JN5121.

Note that Tick Timer interrupts can be used to wake the CPU from doze mode on the JN5148 device, but not on the JN5139 and JN5121 devices.

Parameters

bIntEnable	Enable/disable interrupts: TRUE to enable interrupts FALSE to disable interrupts

Returns

None

bAHI_TickTimerIntStatus

```
bool_t bAHI_TickTimerIntStatus(void);
```

Description

This function obtains the current interrupt status of the Tick Timer.

Parameters

None

Returns

TRUE if an interrupt is pending, FALSE otherwise

vAHI_TickTimerIntPendClr

```
void vAHI_TickTimerIntPendClr(void);
```

Description

This function clears any pending Tick Timer interrupt.

Parameters

None

Returns

None

vAHI_TickTimerInit (JN5139/JN5121 Only)

```
void vAHI_TickTimerInit(
            PR_HWINT_APPCALLBACK prTickTimerCallback);
```

Description

This function registers a user-defined callback function that will be called on a JN5139/JN5121 device when the Tick Timer interrupt is triggered.

Note that the callback function will be executed in interrupt context. You must therefore ensure that it returns to the main program in a timely manner.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A.

Note that the equivalent function for JN5148 is **vAHI_TickTimerRegisterCallback()**.

Parameters

prTickTimerCallback	Pointer to callback function to be registered

Returns

None

vAHI_TickTimerRegisterCallback (JN5148 Only)

void vAHI_TickTimerRegisterCallback(
PR_HWINT_APPCALLBACK *prTickTimerCallback***);**

Description

This function registers a user-defined callback function that will be called on the JN5148 device when the Tick Timer interrupt is triggered.

Note that the callback function will be executed in interrupt context. You must therefore ensure that it returns to the main program in a timely manner.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A.

Note that the equivalent function for JN5139 and JN5121 is **vAHI_TickTimerInit()**.

Parameters

prTickTimerCallback	Pointer to callback function to be registered

Returns

None

10. Watchdog Timer (JN5148 Only)

This chapter describes the functions for configuring and controlling the watchdog timer on the Jennic JN5148 wireless microcontroller.

The watchdog timer implements a timeout to prevent software lock-ups. This timer should be regularly reset (to the start of the timeout period) by the application, in order to prevent the timer from expiring and to indicate that the application still has control of the JN5148 device. If the timer is allowed to expire, the assumption is that the application has lost control of the chip and, thus, a hardware reset of the chip is automatically initiated.

The watchdog timer continues to run during doze mode but not during sleep or deep sleep mode, or when the hardware debugger has taken control of the CPU (it will, however, automatically restart when the debugger un-stalls the CPU).

Following a power-up, reset or wake-up from sleep, the watchdog timer is always enabled with the maximum timer period (regardless of its state before any sleep or reset).

The watchdog timer functions are listed below, along with their page references:

Function	**Page**
vAHI_WatchdogStart (JN5148 Only)	172
vAHI_WatchdogStop (JN5148 Only)	173
vAHI_WatchdogRestart (JN5148 Only)	174
u16AHI_WatchdogReadValue (JN5148 Only)	175
bAHI_WatchdogResetEvent (JN5148 Only)	176

Note: For guidance on using these functions in JN5148 application code, refer to Chapter 10 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

vAHI_WatchdogStart (JN5148 Only)

void vAHI_WatchdogStart(uint8 *u8Prescale***);**

Description

This function starts the watchdog timer and sets the timeout period. Note that the watchdog timer is enabled by default on the JN5148 device and is run with the maximum possible timeout period of 16392 ms. If this function is called while the watchdog timer is running, it allows the timer to continue uninterrupted but modifies the timeout period.

The timeout period of the watchdog timer is determined by an index, specified through the parameter *u8Prescale*, and is calculated according to the formulae:

Timeout Period = 8 ms	if *u8Prescale* = 0
Timeout Period = $[2^{(Prescale\ -\ 1)} + 1] \times 8$ ms	if $1 \leq$ *u8Prescale* ≤ 12

The actual timeout period obtained may be up to 30% less than the calculated value due to variations in the 32-kHz RC oscillator.

Note that the watchdog timer will continue to run during doze mode but not during sleep or deep sleep mode, or when the hardware debugger has taken control of the CPU (it will, however, automatically restart using the same prescale value when the debugger un-stalls the CPU).

Parameters

u8Prescale	Index in the range 0 to 12, which determines the watchdog timeout period (see above formulae) - gives timeout periods in the range 8 to 16392 ms

Returns

None

vAHI_WatchdogStop (JN5148 Only)

```
void vAHI_WatchdogStop(void);
```

Description

This function stops the watchdog timer and freezes the timer count.

Parameters

None

Returns

None

vAHI_WatchdogRestart (JN5148 Only)

```
void vAHI_WatchdogRestart(void);
```

Description

This function re-starts the watchdog timer from the beginning of the timeout period.

Parameters

None

Returns

None

u16AHI_WatchdogReadValue (JN5148 Only)

```
uint16 u16AHI_WatchdogReadValue(void);
```

Description

This function obtains an indication of the progress of the watchdog timer towards its timeout period.

The returned value is an integer in the range 0 to 255, where:

- 0 indicates that the timer has just started a new count
- 255 indicates that the timer has almost reached the timeout period

Thus, each increment of the returned value represents 1/256 of the watchdog period - for example, a reported value of 128 indicates that the timer is about half-way through its count.

If this function is called on a transition (increment) of the watchdog counter, the result will be unreliable. You are therefore advised to call this function repeatedly until two consecutive results are the same.

Tip: This function is useful during code development and debug to ensure that the application does not reset the watchdog timer too close to the watchdog timeout period. The function should not be needed in the final application.

Parameters

None

Returns

Integer value in the range 0 to 255, indicating the progress of the watchdog timer

bAHI_WatchdogResetEvent (JN5148 Only)

```
bool_t bAHI_WatchdogResetEvent(void);
```

Description

This function determines whether the last device reset was caused by a watchdog timer expiry event.

Parameters

None

Returns

TRUE if a reset occurred due to a watchdog event, FALSE otherwise

11. Pulse Counters (JN5148 Only)

This chapter details the functions for controlling and monitoring the pulse counters on the JN5148 device. A pulse counter detects and counts pulses on an external signal that is input on an associated DIO pin.

Two 16-bit pulse counters are provided on the JN5148 device, Pulse Counter 0 and Pulse Counter 1, which receive their input signals on DIO1 and DIO8, respectively. The two counters can be combined together to provide a single 32-bit counter, if desired, in which case the input signal is taken from DIO1.

The pulse counters can increment during all modes of operation, including sleep, and can operate with input signals of up to 100 kHz. An increment can be configured to occur on a rising or falling edge of the relevant DIO input. Each pulse counter has an associated 16-bit reference value that is specified by the user. An interrupt (or wake-up event, if asleep) can be generated when the counter passes this reference value. The counters do not saturate at their maximum count values, but wrap around to 0.

Note: Pulse counter interrupts are handled by the callback function for the System Controller interrupts, registered using **vAHI_SysCtrlRegisterCallback()**.

The pulses can be debounced using the 32-kHz clock, to avoid false counts on slow or noisy edges. With debounce enabled, the maximum possible frequency of the input signal is significantly reduced, depending on the debounce setting. When using debounce, the 32-kHz clock must be active - therefore, for minimum sleep current, the debounce mode should not be used.

The pulse counter functions are listed below, along with their page references:

Function	Page
bAHI_PulseCounterConfigure (JN5148 Only)	178
bAHI_SetPulseCounterRef (JN5148 Only)	180
bAHI_StartPulseCounter (JN5148 Only)	181
bAHI_StopPulseCounter (JN5148 Only)	182
u32AHI_PulseCounterStatus (JN5148 Only)	183
bAHI_Read16BitCounter (JN5148 Only)	184
bAHI_Read32BitCounter (JN5148 Only)	185
bAHI_Clear16BitPulseCounter (JN5148 Only)	186
bAHI_Clear32BitPulseCounter (JN5148 Only)	187

Note: For guidance on using these functions in JN5148 application code, refer to Chapter 11 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

bAHI_PulseCounterConfigure (JN5148 Only)

```
bool_t bAHI_PulseCounterConfigure(uint8 u8Counter,
                                  bool_t bEdgeType,
                                  uint8 u8Debounce,
                                  bool_t bCombine,
                                  bool_t bIntEnable);
```

Description

This function configures the specified pulse counter on the JN5148 device. The input signal will automatically be taken from the DIO associated with the specified counter: DIO1 for Pulse Counter 0 and DIO8 for Pulse Counter 1. The following features are configured:

- **Edge detected** (*bEdgeType*): The counter can be configured to detect a pulse on its rising edge (low-to-high transition) or falling edge (high-to-low transition).
- **Debounce** (*u8Debounce*): This feature can be enabled so that a number of identical consecutive input samples are required before a change in the input signal is recognised. When disabled, the device can sleep with the 32-kHz oscillator off.
- **Combined counter** (*bCombine*): The two 16-bit pulse counters can be combined into a single 32-bit pulse counter. The combined counter is configured according to the Pulse Counter 0 settings (the Pulse Counter 1 settings are ignored) and the input signal is taken from DIO1.
- **Interrupts** (*bIntEnable*): Interrupts can be configured to occur when the count reaches a reference value, specified using **bAHI_SetPulseCounterRef()**. These interrupts are handled as System Controller interrupts by the callback function registered with **vAHI_SysCtrlRegisterCallback()** - also refer to Appendix A.

Parameters

u8Counter	Identity of pulse counter: E_AHI_PC_0 (Pulse Counter 0 or combined counter) E_AHI_PC_1 (Pulse Counter 1)
bEdgeType	Edge type on which pulse detected (and count incremented): 0: Rising edge (low-to-high transition) 1: Falling edge (high-to-low transition)
u8Debounce	Debounce setting - number of identical consecutive input samples before change in input value is recognised: 0: No debounce (maximum input frequency of 100 kHz) 1: 2 samples (maximum input frequency of 3.7 kHz) 2: 4 samples (maximum input frequency of 2.2 kHz) 3: 8 samples (maximum input frequency of 1.2 kHz)
bCombine	Enable/disable combined 32-bit counter: TRUE - Enable combined counter (also set *u8Counter* to E_AHI_PC_0) FALSE - Disable combined counter (use separate counters)
bIntEnable	Enable/disable pulse counter interrupts: TRUE - Enable interrupts FALSE - Disable interrrupts

Returns

TRUE if valid pulse counter specified, FALSE otherwise

bAHI_SetPulseCounterRef (JN5148 Only)

```
bool_t bAHI_SetPulseCounterRef(uint8 u8Counter,
                               uint32 u32RefValue);
```

Description

This function can be used to set the reference value for the specified counter.

If pulse counter interrupts are enabled through **bAHI_PulseCounterConfigure()**, an interrupt will be generated when the counter passes the reference value. This value is retained during sleep and, when generated, the pulse counter interrupt can wake the device from sleep.

The reference value must be 16-bit when specified for the individual pulse counters, but can be a 32-bit value when specified for the combined counter (enabled through **bAHI_PulseCounterConfigure()**). The reference value can be modified at any time.

The pulse counter can increment beyond its reference value and when it reaches its maximum value (65535, or 4294967295 for the combined counter), it will wrap around to zero.

Parameters

u8Counter	Identity of pulse counter: E_AHI_PC_0 (Pulse Counter 0 or combined counter) E_AHI_PC_1 (Pulse Counter 1)
u32RefValue	Reference value to be set - as a 16-bit value, it must be specified in the lower 16 bits of this 32-bit parameter, unless for the combined counter when a full 32-bit value should be specified

Returns

TRUE if valid pulse counter and reference count

FALSE if invalid pulse counter or reference count (>16 bits for single counter)

bAHI_StartPulseCounter (JN5148 Only)

```
bool_t bAHI_StartPulseCounter(uint8 u8Counter);
```

Description

This function starts the specified pulse counter.

Note that the count may increment by one when this function is called (even though no pulse has been detected).

Parameters

u8Counter	Identity of pulse counter: E_AHI_PC_0 (Pulse Counter 0 or combined counter) E_AHI_PC_1 (Pulse Counter 1)

Returns

TRUE if valid pulse counter has been specified and started, FALSE otherwise

bAHI_StopPulseCounter (JN5148 Only)

bool_t bAHI_StopPulseCounter(uint8 *u8Counter***);**

Description

This function stops the specified pulse counter.

Note that the count will freeze when this function is called. Thus, this count can subsequently be read using **bAHI_Read16BitCounter()** or **bAHI_Read32BitCounter()** for the combined counter.

Parameters

u8Counter	Identity of pulse counter: E_AHI_PC_0 (Pulse Counter 0 or combined counter) E_AHI_PC_1 (Pulse Counter 1)

Returns

TRUE if valid pulse counter has been specified and stopped, FALSE otherwise

u32AHI_PulseCounterStatus (JN5148 Only)

```
uint32 u32AHI_PulseCounterStatus(void);
```

Description

This function obtains the status of the pulse counters on the JN5148 device. It can be used to check whether the pulse counters have reached their reference values (set using the function **bAHI_SetPulseCounterRef()**).

The status of each pulse counter is returned by this function in a 32-bit bitmap value - bit 22 for Pulse Counter 0 and bit 23 for Pulse Counter 1. If the combined pulse counter is in use, its status is returned through bit 22.

If a pulse counter has reached its reference value then once the function has returned this status, the internal status bit is cleared for the corresponding pulse counter.

The function can be used to poll the pulse counters. Alternatively, interrupts can be enabled (through **bAHI_PulseCounterConfigure()**) that are generated when the pulse counters reach their reference values.

Parameters

None

Returns

32-bit value in which bit 23 indicates the status of Pulse Counter 1 and bit 22 indicates the status of Pulse Counter 0 or the combined counter. The bit values are interpreted as follows:

1 - pulse counter has reached its reference value
0 - pulse counter is still counting or is not in use

bAHI_Read16BitCounter (JN5148 Only)

```
bool_t bAHI_Read16BitCounter(uint8 u8Counter,
                             uint16 *pu16Count);
```

Description

This function obtains the current count of the specified 16-bit pulse counter, without stopping the counter or clearing the count.

Note that this function can only be used to read the value of an individual 16-bit counter (Pulse Counter 0 or Pulse Counter 1) and cannot read the value of the combined 32-bit counter. If the combined counter is in use, its count value can be obtained using the function **bAHI_Read32BitCounter()**.

Parameters

u8Counter	Identity of pulse counter: E_AHI_PC_0 (Pulse Counter 0) E_AHI_PC_1 (Pulse Counter 1)
**pu16Count*	Pointer to location to receive 16-bit count

Returns

TRUE if valid pulse counter specified, FALSE otherwise

bAHI_Read32BitCounter (JN5148 Only)

```
bool_t bAHI_Read32BitCounter(uint32 *pu32Count);
```

Description

This function obtains the current count of the combined 32-bit pulse counter, without stopping the counter or clearing the count.

Note that this function can only be used to read the value of the combined 32-bit pulse counter and cannot read the value of a 16-bit pulse counter used in isolation. The returned Boolean value of this function indicates if the pulse counters have been combined. If the combined counter is not use, the count value of an individual 16-bit pulse counter can be obtained using the function **bAHI_Read16BitCounter()**.

Parameters

**pu32Count*	Pointer to location to receive 32-bit count

Returns

TRUE if combined 32-bit counter in use, FALSE otherwise

bAHI_Clear16BitPulseCounter (JN5148 Only)

bool_t bAHI_Clear16BitPulseCounter(uint8 const *u8Counter***);**

Description

This function clears the count of the specified 16-bit pulse counter.

Note that this function can only be used to clear the count of an individual 16-bit counter (Pulse Counter 0 or Pulse Counter 1) and cannot clear the count of the combined 32-bit counter. To clear the latter, use the function **bAHI_Clear32BitPulseCounter()**.

Parameters

u8Counter	Identity of pulse counter: E_AHI_PC_0 (Pulse Counter 0) E_AHI_PC_1 (Pulse Counter 1)

Returns

TRUE if valid pulse counter specified, FALSE otherwise

bAHI_Clear32BitPulseCounter (JN5148 Only)

```
bool_t bAHI_Clear32BitPulseCounter(void);
```

Description

This function clears the count of the combined 32-bit pulse counter.

Note that this function can only be used to clear the count of the combined 32-bit pulse counter and cannot clear the count of a 16-bit pulse counter used in isolation. To clear the latter, use the function **bAHI_Clear16BitPulseCounter()**.

Parameters

None

Returns

TRUE if combined 32-bit counter in use, FALSE otherwise

12. Serial Interface (2-wire)

This chapter details the functions for controlling the 2-wire Serial Interface (SI) on Jennic wireless microcontrollers. The Serial Interface is logic-compatible with similar interfaces such as I^2C and SMbus.

Two sets of functions are described in this chapter, one set for an SI master and another set for an SI slave:

- An SI master is a feature of all the Jennic wireless microcontrollers and functions for controlling the SI master are described in Section 12.1.
- An SI slave is provided only on the JN5148 device and the functions for controlling the SI slave are described in Section 12.2.

Tip: The protocol used by the Serial Interface is detailed in the I^2C Specification (available from www.nxp.com).

Note: For guidance on using the SI functions in application code, refer to Chapter 12 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

12.1 SI Master Functions

This section details the functions for controlling a 2-wire Serial Interface (SI) master on a Jennic wireless microcontroller.

The SI master can implement bi-directional communication with a slave device on the SI bus (SI slave functions are provided for the JN5148 device and are described in Section 12.2). Note that the SI bus on the JN5148 device can have more than one master, but multiple masters cannot use the bus at the same time - to avoid this, an arbitration scheme is provided.

When enabled, this interface uses DIO14 as a clock and DIO15 as a bi-directional data line. The clock is scaled from the 16-MHz system clock.

The SI master functions are listed below, along with their page references:

Function	**Page**
vAHI_SiConfigure (JN5139 Only)	191
vAHI_SiMasterConfigure (JN5148 Only)	192
vAHI_SiMasterDisable (JN5148 Only)	193
bAHI_SiMasterSetCmdReg	194
vAHI_SiMasterWriteSlaveAddr	196
vAHI_SiMasterWriteData8	197
u8AHI_SiMasterReadData8	198
bAHI_SiMasterPollBusy	199
bAHI_SiMasterPollTransferInProgress	200
bAHI_SiMasterCheckRxNack	201
bAHI_SiMasterPollArbitrationLost	202
vAHI_SiRegisterCallback	203

Note that the SI function set in earlier releases of this API comprised a subset of the above functions with slightly different names (the word ‘Master’ was omitted). These old names are still valid (they are aliased to the new functions) and are as follows:

vAHI_SiSetCmdReg
vAHI_SiWriteData8
vAHI_SiWriteSlaveAddr
u8AHI_SiReadData8
bAHI_SiPollBusy
bAHI_SiPollTransferInProgress
bAHI_SiPollRxNack (previously **bAHI_SiCheckRxNack**)
bAHI_SiPollArbitrationLost

vAHI_SiConfigure (JN5139 Only)

```
void vAHI_SiConfigure(bool_t bSiEnable,
                      bool_t bInterruptEnable,
                      uint16 u16PreScaler);
```

Description

This function is used to enable/disable and configure the 2-wire Serial Interface (SI) master on the JN5139 device. This function must be called to enable the SI block before any other SI master function is called.

The operating frequency, derived from the 16-MHz system clock using the specified prescaler *u16PreScaler*, is given by:

Operating frequency = 16/[(*PreScaler* + 1) x 5] MHz

The prescaler is a 16-bit value for the JN5139 device.

Parameters

bSiEnable	Enable/disable Serial Interface master: TRUE - enable FALSE - disable
bInterruptEnable	Enable/disable Serial Interface interrupt: TRUE - enable FALSE - disable
u16PreScaler	16-bit clock prescaler (see above)

Returns

None

vAHI_SiMasterConfigure (JN5148 Only)

```
void vAHI_SiMasterConfigure(
                                  bool_t bPulseSuppressionEnable,
                                  bool_t bInterruptEnable,
                                  uint8 u8PreScaler);
```

Description

This function is used to configure and enable the 2-wire Serial Interface (SI) master on the JN5148 device. This function must be called to enable the SI block before any other SI master function is called. To later disable the interface, the function **vAHI_SiMasterDisable()** must be used.

The operating frequency, derived from the 16-MHz system clock using the specified prescaler *u8PreScaler*, is given by:

Operating frequency = 16/[(*PreScaler* + 1) x 5] MHz

The prescaler is an 8-bit value for the JN5148 device.

A pulse suppression filter can be enabled to suppress any spurious pulses (high or low) with a pulse width less than 62.5 ns on the clock and data lines.

Parameters

bPulseSuppressionEnable	Enable/disable pulse suppression filter: TRUE - enable FALSE - disable
bInterruptEnable	Enable/disable Serial Interface interrupt: TRUE - enable FALSE - disable
u8PreScaler	8-bit clock prescaler (see above)

Returns

None

vAHI_SiMasterDisable (JN5148 Only)

```
void vAHI_SiMasterDisable(void);
```

Description

This function disables (and powers down) the SI master on the JN5148 device, if it has been previously enabled using the function **vAHI_SiMasterConfigure()**.

Parameters

None

Returns

None

bAHI_SiMasterSetCmdReg

```
bool_t bAHI_SiMasterSetCmdReg(bool_t bSetSTA,
                              bool_t bSetSTO,
                              bool_t bSetRD,
                              bool_t bSetWR,
                              bool_t bSetAckCtrl,
                              bool_t bSetIACK);
```

Description

This function configures the combination of I^2C-protocol commands for a transfer on the SI bus and starts the transfer of the data held in the SI master's transmit buffer.

Up to four commands can be used to perform an I^2C-protocol transfer - Start, Stop, Write, Read. This function allows these commands to be combined to form a complete or partial transfer sequence. The valid command combinations that can be specified are summarised below.

Start	Stop	Read	Write	Resulting Instruction to SI Bus
0	0	0	0	No active command (idle)
1	0	0	1	Start followed by Write
1	1	0	1	Start followed by Write followed by Stop
0	1	1	0	Read followed by Stop
0	1	0	1	Write followed by Stop
0	0	0	1	Write only
0	0	1	0	Read only
0	1	0	0	Stop only

The above command combinations will result in the function returning TRUE, while command combinations that are not in the above list are invalid and will result in a FALSE return code.

The function must be called immediately after **vAHI_SiMasterWriteSlaveAddr()**, which puts the destination slave address (for the subsequent data transfer) into the transmit buffer. It must then be called immediately after **vAHI_SiMasterWriteData()** to start the transfer of data (from the transmit buffer).

To implement a data transfer on the SI bus, you must follow the process described in the I^2C Specification.

Note: This function replaces **vAHI_SiMasterSetCmdReg()**, which returns no value. However, the previous function is still available in the API for backward compatibility.

Parameters

bSetSTA	Generate START bit to gain control of the SI bus (must not be enabled with STOP bit): E_AHI_SI_START_BIT E_AHI_SI_NO_START_BIT
bSetSTO	Generate STOP bit to release control of the SI bus (must not be enabled with START bit): E_AHI_SI_STOP_BIT E_AHI_SI_NO_STOP_BIT
bSetRD	Read from slave (cannot be enabled with slave write): E_AHI_SI_SLAVE_READ E_AHI_SI_NO_SLAVE_READ
bSetWR	Write to slave (cannot be enabled with slave read): E_AHI_SI_SLAVE_WRITE E_AHI_SI_NO_SLAVE_WRITE
bSetAckCtrl	Send ACK or NACK to slave after each byte read: E_AHI_SI_SEND_ACK (to indicate ready for next byte) E_AHI_SI_SEND_NACK (to indicate no more data required)
bSetIACK	Generate interrupt acknowledge (set to clear pending interrupt): E_AHI_SI_IRQ_ACK E_AHI_SI_NO_IRQ_ACK

Returns

TRUE if specified command combination is legal
FALSE if specified command combination is illegal (will result in no action by device)

vAHI_SiMasterWriteSlaveAddr

```
void vAHI_SiMasterWriteSlaveAddr(uint8 u8SlaveAddress,
                                 bool_t bReadStatus);
```

Description

This function is used in setting up communication with a slave device. In this function, you must specify the address of the slave (see below) and the operation (read or write) to be performed on the slave. The function puts this information in the SI master's transmit buffer, but the information will be not transmitted on the SI bus until the function **bAHI_SiMasterSetCmdReg()** is called.

A slave address can be 7-bit or 10-bit, where this address size is set using the function **vAHI_SiSlaveConfigure()** called on the slave device. **vAHI_SiMasterWriteSlaveAddr()** is used differently for the two slave addressing modes:

- For 7-bit addressing, the parameter *u8SlaveAddress* must be set to the 7-bit slave address.
- For 10-bit addressing, the parameter *u8SlaveAddress* must be set to the binary value 011110xx, where xx are the 2 most significant bits of the 10-bit slave address - the code 011110 indicates to the SI bus slaves that 10-bit addressing will be used in the next communication. The remaining 8 bits of the slave address must subsequently be specified in a call to **vAHI_SiMasterWriteData8()**.

To implement a data transfer on the SI bus, you must follow the process described in the I^{2}C Specification.

Parameters

u8SlaveAddress	Slave address (see above)
bReadStatus	Operation to perform on slave (read or write): TRUE - configure a read FALSE - configure a write

Returns

None

vAHI_SiMasterWriteData8

void vAHI_SiMasterWriteData8(uint8 *u8Out***);**

Description

This function writes a single data-byte to the transmit buffer of the SI master.

The contents of the transmit buffer will not be transmitted on the SI bus until the function **bAHI_SiMasterSetCmdReg()** is called.

Parameters

u8Out	8 bits of data to transmit

Returns

None

u8AHI_SiMasterReadData8

```
uint8 u8AHI_SiMasterReadData8(void);
```

Description

This function obtains a data-byte received over the SI bus.

Parameters

None

Returns

Data read from receive buffer of SI master

bAHI_SiMasterPollBusy

```
bool_t bAHI_SiMasterPollBusy(void);
```

Description

This function checks whether the SI bus is busy (could be in use by another master).

Parameters

None

Returns

TRUE if busy, FALSE otherwise

bAHI_SiMasterPollTransferInProgress

```
bool_t bAHI_SiMasterPollTransferInProgress(void);
```

Description

This function checks whether a transfer is in progress on the SI bus.

Parameters

None

Returns

TRUE if a transfer is in progress, FALSE otherwise

bAHI_SiMasterCheckRxNack

```
bool_t bAHI_SiMasterCheckRxNack(void);
```

Description

This function checks whether a NACK or an ACK has been received from the slave device. If a NACK has been received, this indicates that the SI master should stop sending data to the slave.

Parameters

None

Returns

TRUE if NACK has occurred

FALSE if ACK has occurred

bAHI_SiMasterPollArbitrationLost

```
bool_t bAHI_SiMasterPollArbitrationLost(void);
```

Description

This function checks whether arbitration has been lost (by the local master) on the SI bus.

Parameters

None

Returns

TRUE if arbitration loss has occurred, FALSE otherwise

vAHI_SiRegisterCallback

**void vAHI_SiRegisterCallback(
PR_HWINT_APPCALLBACK** *prSiCallback***);**

Description

This function registers a user-defined callback function that will be called when a Serial Interface interrupt is triggered on the SI master.

Note that this function can be used to register the callback function for a SI slave as well as for the SI master. The SI interrupt handler will determine whether a SI interrupt has been generated on a master or slave, and then invoke the relevant callback function.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A.

Parameters

prSiCallback	Pointer to callback function to be registered

Returns

None

12.2 SI Slave Functions (JN5148 Only)

This section details the functions for controlling a 2-wire Serial Interface (SI) slave on the Jennic JN5148 wireless microcontroller.

The SI slave functions are listed below, along with their page references:

Function	**Page**
vAHI_SiSlaveConfigure (JN5148 Only)	205
vAHI_SiSlaveDisable (JN5148 Only)	207
vAHI_SiSlaveWriteData8 (JN5148 Only)	208
u8AHI_SiSlaveReadData8 (JN5148 Only)	209
vAHI_SiRegisterCallback	210

vAHI_SiSlaveConfigure (JN5148 Only)

```
void vAHI_SiSlaveConfigure(
                    uint16 u16SlaveAddress,
                    bool_t bExtendAddr,
                    bool_t bPulseSuppressionEnable,
                    uint8 u8InMaskEnable,
                    bool_t bFlowCtrlMode);
```

Description

This function is used to configure and enable the 2-wire Serial Interface (SI) slave on the JN5148 device. This function must be called before any other SI slave function. To later disable the interface, the function **vAHI_SiSlaveDisable()** must be used.

You must specify the address of the slave to be configured and enabled. A 7-bit or 10-bit slave address can be used. The address size must also be specified through *bExtendAddr*.

The function allows SI slave interrupts to be enabled on an individual basis using an 8-bit bitmask specified through *u8InMaskEnable*. The SI slave interrupts are enumerated as follows:

Bit	Enumeration	Interrupt Description
0	E_AHI_SIS_DATA_RR_MASK	Data buffer must be written with data to be read by SI master
1	E_AHI_SIS_DATA_RTKN_MASK	Data taken from buffer by SI master - buffer free for next data
2	E_AHI_SIS_DATA_WA_MASK	Data buffer contains data from SI master to be read by SI slave
3	E_AHI_SIS_LAST_DATA_MASK	Last data transferred (end of burst)
4	E_AHI_SIS_ERROR_MASK	I^{2}C protocol error

To obtain the bitmask for *u8InMaskEnable*, the enumerations for the interrupts to be enabled can be ORed together.

A pulse suppression filter can be enabled to suppress any spurious pulses (high or low) with a pulse width less than 62.5 ns on the clock and data lines.

Parameters

u16SlaveAddress	Slave address (7-bit or 10-bit, as defined by *bExtendAdd*)
bExtendAddr	Size of slave address (specified through *u16SlaveAddress*): TRUE - 10-bit address FALSE - 7-bit address

bPulseSuppressionEnable	Enable/disable pulse suppression filter: TRUE - enable FALSE - disable
u8InMaskEnable	Bitmask of SI slave interrupts to be enabled (see above)
bFlowCtrlMode	Flow control mode: TRUE - use clock stretching to hold bus until space available to write data FALSE - use NACK (default)

Returns

None

vAHI_SiSlaveDisable (JN5148 Only)

```
void vAHI_SiSlaveDisable(void);
```

Description

This function disables (and powers down) the SI slave on the JN5148 device, if it has been previously enabled using the function **vAHI_SiSlaveConfigure()**.

Parameters

None

Returns

None

vAHI_SiSlaveWriteData8 (JN5148 Only)

void vAHI_SiSlaveWriteData8(uint8 *u8Out***);**

Description

This function writes a single byte of output data to the data buffer of the SI slave on the JN5148 device, ready to be read by the SI master.

Parameters

u8Out	8 bits of output data

Returns

None

u8AHI_SiSlaveReadData8 (JN5148 Only)

```
uint8 u8AHI_SiSlaveReadData8(void);
```

Description

This function reads a single byte of input data from the buffer of the SI slave on the JN5148 device (where this data byte has been received from the SI master).

Parameters

None

Returns

Input data-byte read from buffer of SI slave

vAHI_SiRegisterCallback

```
void vAHI_SiRegisterCallback(
                    PR_HWINT_APPCALLBACK prSiCallback);
```

Description

This function registers a user-defined callback function that will be called when a Serial Interface interrupt is triggered on a SI slave.

Note that this function can be used to register the callback function for the SI master as well as for a SI slave. The SI interrupt handler will determine whether a SI interrupt has been generated on a master or slave, and then invoke the relevant callback function.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A.

Parameters

prSiCallback	Pointer to callback function to be registered

Returns

None

13. Serial Peripheral Interface (SPI Master)

This chapter details the functions for controlling the Serial Peripheral Interface (SPI) on the Jennic wireless microcontrollers. The SPI allows high-speed synchronous data transfer between the microcontroller and peripheral devices. The microcontroller operates as a master on the SPI bus and all other devices connected to the SPI are expected to be slave devices under the control of the microcontroller's CPU.

The SPI master can be used to communicate with up to five attached peripherals, including the Flash memory. It can transfer 8, 16 or 32 bits without software intervention and can keep the slave-select lines asserted between transfers, when required, to allow longer transfers to be performed.

As well as dedicated pins for Data In, Data Out, Clock and Slave Select 0, the SPI master can be configured to enable up to 4 more slave-select lines which appear on DIO0 to DIO3. Slave-select 0 is assumed to be connected to Flash memory and is read during the boot sequence.

The SPI master functions are listed below, along with their page references:

Function	**Page**
vAHI_SpiConfigure (JN5121 Version)	212
vAHI_SpiConfigure (JN5139/JN5148 Version)	214
vAHI_SpiReadConfiguration	216
vAHI_SpiRestoreConfiguration	217
vAHI_SpiSelect	218
vAHI_SpiStop	219
vAHI_SpiStartTransfer32 (JN5139/JN5121 Only)	221
u32AHI_SpiReadTransfer32	222
vAHI_SpiStartTransfer16 (JN5139/JN5121 Only)	223
u16AHI_SpiReadTransfer16	224
vAHI_SpiStartTransfer8 (JN5139/JN5121 Only)	225
u8AHI_SpiReadTransfer8	226
vAHI_SpiContinuous (JN5148 Only)	227
bAHI_SpiPollBusy	228
vAHI_SpiWaitBusy	229
vAHI_SetDelayReadEdge (JN5148 Only)	230
vAHI_SpiRegisterCallback	231

Note 1: For guidance on using the SPI master functions in application code, refer to Chapter 13 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

Note 2: SPI slave functions are detailed in Chapter 14.

vAHI_SpiConfigure (JN5121 Version)

```
void vAHI_SpiConfigure(uint8 u8SlaveEnable,
                       bool_t bLsbFirst,
                       bool_t bTxPosEdge,
                       bool_t bRxNegEdge,
                       uint8 u8ClockDivider,
                       bool_t bInterruptEnable,
                       bool_t bAutoSlaveSelect);
```

Description

This function configures and enables the SPI master on the JN5121 device.

By default, there is one SPI slave (the Flash memory) with a dedicated IO pin for its select line. Depending on how many additional slaves are enabled, up to four more select lines can be set, which use DIO pins 0 to 3. For example, if two additional slaves are enabled, DIO 0 and 1 will be assigned. Note that once reserved for SPI use, DIO lines cannot be subsequently released by calling this function again (and specifying a smaller number of SPI slaves).

The following features are also configurable using this function:

- Data transfer order - whether the least significant bit is transferred first or last
- Transmit clock edge - transmit data to change on the positive edge of the clock and therefore to be sampled by the slave device on the negative edge of the clock, or vice versa
- Receive clock edge - received data to be sampled on the negative edge of the clock and therefore to be changed by the slave device on the positive edge, or vice versa
- Clock divisor - the value used to derive the SPI clock from the 16-MHz system clock
- SPI interrupt - generated when an API transfer has completed (note that interrupts are only worth using if the SPI clock frequency is much less than 16 MHz)
- Automatic slave selection - enable the programmed slave select line or lines (see **vAHI_SpiSelect()**) to be automatically asserted at the start of a transfer and de-asserted when the transfer completes. If not enabled, the slave select lines will reflect the value set by **vAHI_SpiSelect()** directly.

Parameters

u8SlaveEnable	Number of extra SPI slaves to control. Valid values are 0 to 4 - higher values are truncated to 4
bLsbFirst	Enable/disable data transfer with the least significant bit (LSB) transferred first: TRUE - enable FALSE - disable
bTxPosEdge	Transmit clock edge: TRUE - transmit data to change on positive clock edge and be sampled by slave device on negative clock edge FALSE - transmit data to change on negative clock edge and be sampled by slave device on positive clock edge

bRxNegEdge	Receive clock edge: TRUE - received data to be sampled on negative clock edge and changed by slave device on positive clock edge FALSE - received data to be sampled on positive clock edge and changed by slave device on negative clock edge
u8ClockDivider	Clock divisor in the range 0 to 63 - 16-MHz clock is divided by 2 x *u8ClockDivider,* but 0 is a special value used when no clock division is required (to obtain a 16-MHz SPI bus clock)
bInterruptEnable	Enable/disable interrupt when an SPI transfer has completed: TRUE - enable FALSE - disable
bAutoSlaveSelect	Enable/disable automatic slave selection: TRUE - enable FALSE - disable

Returns

None

vAHI_SpiConfigure (JN5139/JN5148 Version)

void vAHI_SpiConfigure(uint8 *u8SlaveEnable*,
bool_t *bLsbFirst*,
bool_t *bPolarity*,
bool_t *bPhase*,
uint8 *u8ClockDivider*,
bool_t *bInterruptEnable*,
bool_t *bAutoSlaveSelect***);**

Description

This function configures and enables the SPI master on the JN5139/JN5148 device.

The function allows the number of extra SPI slaves (of the master) to be set. By default, there is one SPI slave (the Flash memory) with a dedicated IO pin for its select line. Depending on how many additional slaves are enabled, up to four more select lines can be set, which use DIO pins 0 to 3. For example, if two additional slaves are enabled, DIO 0 and 1 will be assigned. Note that once reserved for SPI use, DIO lines cannot be subsequently released by calling this function again (and specifying a smaller number of SPI slaves).

The following features are also configurable using this function:

- Data transfer order - whether the least significant bit is transferred first or last
- Clock polarity and phase, which together determine the SPI mode (0, 1, 2 or 3) and therefore the clock edge on which data is latched:
  - SPI Mode 0: polarity=0, phase=0
  - SPI Mode 1: polarity=0, phase=1
  - SPI Mode 2: polarity=1, phase=0
  - SPI Mode 3: polarity=1, phase=1
- Clock divisor - the value used to derive the SPI clock from the 16-MHz system clock
- SPI interrupt - generated when an API transfer has completed (note that interrupts are only worth using if the SPI clock frequency is much less than 16 MHz)
- Automatic slave selection - enable the programmed slave-select line or lines (see **vAHI_SpiSelect()**) to be automatically asserted at the start of a transfer and de-asserted when the transfer completes. If not enabled, the slave-select lines will reflect the value set by **vAHI_SpiSelect()** directly.

Parameters

u8SlaveEnable	Number of extra SPI slaves to control. Valid values are 0 to 4 - higher values are truncated to 4
bLsbFirst	Enable/disable data transfer with the least significant bit (LSB) transferred first: TRUE - enable FALSE - disable
bPolarity	Clock polarity: FALSE - unchanged TRUE - inverted

bPhase	Phase: FALSE - latch data on leading edge of clock TRUE - latch data on trailing edge of clock
u8ClockDivider	Clock divisor in the range 0 to 63 - 16-MHz clock is divided by 2 x *u8ClockDivider,* but 0 is a special value used when no clock division is required (to obtain a 16-MHz SPI bus clock)
bInterruptEnable	Enable/disable interrupt when an SPI transfer has completed: TRUE - enable FALSE - disable
bAutoSlaveSelect	Enable/disable automatic slave selection: TRUE - enable FALSE - disable

Note that the parameters *bPolarity* and *bPhase* are named differently in the library header file.

Returns

None

vAHI_SpiReadConfiguration

```
void vAHI_SpiReadConfiguration(
                    tSpiConfiguration *ptConfiguration);
```

Description

This function obtains the current configuration of the SPI bus.

This function is intended to be used in a system where the SPI bus is used in multiple configurations to allow the state to be restored later using the function **vAHI_SpiRestoreConfiguration()**. Therefore, no knowledge is needed of the configuration details.

Parameters

**ptConfiguration*	Pointer to location to receive obtained SPI configuration

Returns

None

vAHI_SpiRestoreConfiguration

```
void vAHI_SpiRestoreConfiguration(
                    tSpiConfiguration *ptConfiguration);
```

Description

This function restores the SPI bus configuration using the configuration previously obtained using **vAHI_SpiReadConfiguration()**.

Parameters

**ptConfiguration*	Pointer to SPI configuration to be restored

Returns

None

vAHI_SpiSelect

void vAHI_SpiSelect(uint8 *u8SlaveMask***);**

Description

This function sets the active slave-select line(s) to use.

The slave-select lines are asserted immediately if “automatic slave selection” is disabled, or otherwise only during data transfers. The number of valid bits in *u8SlaveMask* depends on the setting of *u8SlaveEnable* in a previous call to **vAHI_SpiConfigure()**, as follows:

u8SlaveEnable	**Valid bits in *u8SlaveMask***
0	Bit 0
1	Bits 0, 1
2	Bits 0, 1, 2
3	Bits 0, 1, 2, 3
4	Bits 0, 1, 2, 3, 4

Parameters

u8SlaveMask	Bitmap - one bit per slave-select line

Returns

None

vAHI_SpiStop

```
void vAHI_SpiStop(void);
```

Description

This function clears any active slave-select lines. It has the same effect as **vAHI_SpiSelect(0)**.

Parameters

None

Returns

None

vAHI_SpiStartTransfer (JN5148 Only)

void vAHI_SpiStartTransfer(uint8 *u8CharLen***, uint32** *u32Out***);**

Description

This function starts a data transfer to selected slave(s). The data length for the transfer can be specified in the range 1 to 32 bits.

Note: This function can only be used on the JN5148 device. For the JN5139/JN5121 devices, individual functions are provided to start 8-bit, 16-bit and 32-bit transfers.

It is assumed that **vAHI_SpiSelect()** has been called to set the slave(s) to communicate with. If interrupts are enabled for the SPI master, an interrupt will be generated when the transfer has completed.

The function **u32AHI_SpiReadTransfer32()** should be used to read the transferred data, with the data aligned to the right (lower bits).

Parameters

u8CharLen	Value in range 0-31 indicating data length for transfer: 0 - 1-bit data 1 - 2-bit data 2 - 3-bit data : 31 - 32-bit data
u32Out	Data to transmit, aligned to the right (e.g. for an 8-bit transfer, store the data in bits 0-7)

Returns

None

vAHI_SpiStartTransfer32 (JN5139/JN5121 Only)

```
void vAHI_SpiStartTransfer32(uint32 u32Out);
```

Description

This function starts a 32-bit data transfer to selected slave(s). This function can only be used on the JN5139/JN5121 devices - the equivalent function **vAHI_SpiStartTransfer()** must be used on the JN5148 device.

It is assumed that **vAHI_SpiSelect()** has been called to set the slave(s) to communicate with. If interrupts are enabled for the SPI master, an interrupt will be generated when the transfer has completed.

Parameters

u32Out	32 bits of data to transmit

Returns

None

u32AHI_SpiReadTransfer32

```
uint32 u32AHI_SpiReadTransfer32(void);
```

Description

This function obtains the received data after a SPI transfer has completed that was started using **vAHI_SpiStartTransfer32()**, **vAHI_SpiStartTransfer()** or **vAHI_SpiSetContinuous()**. In the cases of the last two functions, the read data is aligned to the right (lower bits).

Parameters

None

Returns

Received data (32 bits)

vAHI_SpiStartTransfer16 (JN5139/JN5121 Only)

```
void vAHI_SpiStartTransfer16(uint16 u16Out);
```

Description

This function starts a 16-bit data transfer to selected slave(s). This function can only be used on the JN5139/JN5121 devices - the equivalent function **vAHI_SpiStartTransfer()** must be used on the JN5148 device.

It is assumed that **vAHI_SpiSelect()** has been called to set the slave(s) to communicate with. If interrupts are enabled for the SPI master, an interrupt will be generated when the transfer has completed.

Parameters

u16Out	16 bits of data to transmit

Returns

None

u16AHI_SpiReadTransfer16

```
uint16 u16AHI_SpiReadTransfer16(void);
```

Description

This function obtains the received data after a 16-bit SPI transfer has completed.

Parameters

None

Returns

Received data (16 bits)

vAHI_SpiStartTransfer8 (JN5139/JN5121 Only)

void vAHI_SpiStartTransfer8(uint8 *u8Out***);**

Description

This function starts an 8-bit transfer to selected slaves(s). This function can only be used on the JN5139/JN5121 devices - the equivalent function **vAHI_SpiStartTransfer()** must be used on the JN5148 device.

It is assumed that **vAHI_SpiSelect()** has been called to set the slave(s) to communicate with. If interrupts are enabled for the SPI master, an interrupt will be generated when the transfer has completed. If interrupts are not enabled for the SPI master, the function **bAHI_SpiPollBusy()** or **vAHI_SpiWaitBusy()** can be used to determine whether the transfer has completed.

Parameters

u8Out	8 bits of data to transmit

Returns

None

u8AHI_SpiReadTransfer8

```
uint8 u8AHI_SpiReadTransfer8(void);
```

Description

This function obtains the received data after a 8-bit SPI transfer has completed.

Parameters

None

Returns

Received data (8 bits)

vAHI_SpiContinuous (JN5148 Only)

```
void vAHI_SpiContinuous(bool_t bEnable,
                        uint8 u8CharLen);
```

Description

This function can be used on the JN5148 device to enable/disable continuous read mode. The function allows continuous data transfers to the SPI master and facilitates back-to-back reads of the received data. In this mode, incoming data transfers are automatically controlled by hardware - data is received and the hardware then waits for this data to be read by the software before allowing the next data transfer.

The data length for an individual transfer can be specified in the range 1 to 32 bits.

If used to enable continuous mode, the function will start the transfers (so there is no need to call a SPI start transfer function. If used to disable continuous mode, the function will stop any existing transfers (following the function call, one more transfer is made before the transfers are stopped).

To determine when data is ready to be read, the application should check whether the interface is busy by calling the function **bAHI_SpiPollBusy()**. If it is not busy receiving data, the data from the previous transfer can be read by calling **u32AHI_SpiReadTransfer32()**, with the data aligned to the right (lower bits). Once the data has been read, the next transfer will automatically occur.

Parameters

bEnable	Enable/disable continuous read mode and start/stop transfers: TRUE - enable mode and start transfers FALSE - stop transfers and disable mode
u8CharLen	Value in range 0-31 indicating data length for transfer: 0 - 1-bit data 1 - 2-bit data 2 - 3-bit data : 31 - 32-bit data

Returns

None

bAHI_SpiPollBusy

```
bool_t bAHI_SpiPollBusy(void);
```

Description

This function polls the SPI master to determine whether it is currently busy performing a data transfer.

Parameters

None

Returns

TRUE if the SPI master is performing a transfer, FALSE otherwise

vAHI_SpiWaitBusy

```
void vAHI_SpiWaitBusy(void);
```

Description

This function waits for the SPI master to complete a transfer and then returns.

Parameters

None

Returns

None

vAHI_SetDelayReadEdge (JN5148 Only)

void vAHI_SpiSetDelayReadEdge(bool_t *bSetDreBit***);**

Description

This function can be used on the JN5148 device to introduce a delay to the SCLK edge used to sample received data. The delay is by half a SCLK period relative to the normal position (so is the sameedge used by the slave device to transmit the next data bit).

The function should be used when the round-trip delay of SCLK out to MISO IN is large compared with half a SCLK period (e.g. fast SCLK, low voltage, slow slave device), to allow a faster transfer rate to be used than would otherwise be possible.

Parameters

bSetDreBit	Enable/disable read edge delay: TRUE - enable FALSE - disable

Returns

None

vAHI_SpiRegisterCallback

```
void vAHI_SpiRegisterCallback(
        PR_HWINT_APPCALLBACK prSpiCallback);
```

Description

This function registers an application callback that will be called when the SPI interrupt is triggered.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A.

Parameters

prSpiCallback	Pointer to callback function to be registered

Returns

None

14. Intelligent Peripheral Interface (SPI Slave)

This chapter details the functions for controlling the Intelligent Peripheral (IP) interface of the Jennic wireless microcontrollers.

The Intelligent Peripheral interface is a SPI (Serial Peripheral Interface) slave and uses pins shared with Digital IO signals DIO14-18. The interface is designed to allow message passing and data transfer. Data received and transmitted on the IP interface is copied directly to and from a dedicated area of memory without intervention from the CPU. This memory area, the Intelligent Peripheral memory block, contains receive and transmit buffers, each comprising sixty-three 32-bit words.

For more details of the data message format, refer to the data sheet for your Jennic wireless microcontroller.

The IP functions are listed below, along with their page references:

Function	Page
vAHI_IpEnable (JN5148 Version)	234
vAHI_IpEnable (JN5139/JN5121 Version)	235
vAHI_IpDisable (JN5148 Only)	236
bAHI_IpSendData (JN5148 Version)	237
bAHI_IpSendData (JN5139/JN5121 Version)	238
bAHI_IpReadData (JN5148 Version)	239
bAHI_IpReadData (JN5139/JN5121 Version)	240
bAHI_IpTxDone	241
bAHI_IpRxDataAvailable	242
vAHI_IpReadyToReceive (JN5148 Only)	243
vAHI_IpRegisterCallback	244

Note 1: For guidance on using the IP (SPI slave) functions in application code, refer to Chapter 14 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

Note: SPI master functions are detailed in Chapter 13.

vAHI_IpEnable (JN5148 Version)

void vAHI_IpEnable(bool_t *bTxEdge*,
bool_t *bRxEdge*,
bool_t *bIntEn***);**

Description

This function initialises and enables the Intelligent Peripheral (IP) interface on the JN5148 device.

The function allows the clock edges to be selected on which receive data will be sampled and transmit data will be changed (but see Caution below). It also allows Intelligent Peripheral interrupts to be enabled/disabled.

Caution: *Only one mode of the IP interface (SPI mode 0) is supported, in which data is transmitted on a negative clock edge and received on a positive clock edge. The parameters bTxEdge and bRxEdge must be set accordingly (both to 0).*

Parameters

bTxEdge	Clock edge that transmit data is changed on (see Caution): E_AHI_IP_TXPOS_EDGE (data changed on +ve edge, to be sampled on -ve edge) E_AHI_IP_TXNEG_EDGE (data changed on -ve edge, to be sampled on +ve edge)
bRxEdge	Clock edge that receive data is sampled on (see Caution): E_AHI_IP_RXPOS_EDGE (data sampled on +ve edge) E_AHI_IP_RXNEG_EDGE (data sampled on -ve edge)
bIntEn	Enable/disable Intelligent Peripheral interrupts: TRUE - enable interrupts FALSE - disable interrupts

Returns

None

vAHI_IpEnable (JN5139/JN5121 Version)

void vAHI_IpEnable(bool_t *bTxEdge*,
bool_t *bRxEdge*,
bool_t *bEndian***);**

Description

This function initialises and enables the Intelligent Peripheral (IP) interface on the JN5139/JN5121 device. Intelligent Peripheral interrupts are also enabled when this function is called.

The function allows the clock edges to be selected on which receive data will be sampled and transmit data will be changed (but see Caution below). It also allows Intelligent Peripheral interrupts to be enabled/disabled.

The function also requires the byte order (Big or Little Endian) of the data for the IP interface to be specified.

Caution: *Only one mode of the IP interface (SPI mode 0) is supported, in which data is transmitted on a negative clock edge and received on a positive clock edge. The parameters bTxEdge and bRxEdge must be set accordingly (both to 0).*

Parameters

bTxEdge	Clock edge that transmit data is changed on (see Caution): E_AHI_IP_TXPOS_EDGE (data changed on +ve edge, to be sampled on -ve edge) E_AHI_IP_TXNEG_EDGE (data changed on -ve edge, to be sampled on +ve edge)
bRxEdge	Clock edge that receive data is sampled on (see Caution): E_AHI_IP_RXPOS_EDGE (data sampled on +ve edge) E_AHI_IP_RXNEG_EDGE (data sampled on -ve edge)
bEndian	Byte order (Big or Little Endian) of data over the IP interface: E_AHI_IP_BIG_ENDIAN E_AHI_IP_LITTLE_ENDIAN

Returns

None

vAHI_IpDisable (JN5148 Only)

```
void vAHI_IpDisable(void);
```

Description

This function disables the Intelligent Peripheral (IP) interface on the JN5148 device.

Parameters

None

Returns

None

bAHI_IpSendData (JN5148 Version)

```
bool_t bAHI_IpSendData(uint8 u8Length,
                       uint8 *pau8Data,
                       bool_t bEndian);
```

Description

This function is used on the JN5148 device to copy data from RAM to the IP Transmit buffer and to indicate that data is ready to be transmitted across the IP interface to the remote processor (the SPI master).

The function requires the data length to be specified, as well as a pointer to a RAM buffer containing the data and the byte order (Big or Little Endian) of the data. The data should be stored in the RAM buffer according to the byte order specified.

The function copies the specified data to the IP Transmit buffer, ready to be sent when the master device initiates the transfer. The IP_INT pin is also asserted to indicate to the master that data is ready to be sent.

The data length is transmitted in the first 32-bit word of the data payload. It is the responsibility of the SPI master receiving the data to retrieve the data length from the payload.

Parameters

u8Length	Length of data to be sent (in 32-bit words)
**pau8Data*	Pointer to RAM buffer containing the data to be sent
bEndian	Byte order (Big or Little Endian) of data over the IP interface: E_AHI_IP_BIG_ENDIAN E_AHI_IP_LITTLE_ENDIAN

Returns

TRUE if successful, FALSE if unable to send

bAHI_IpSendData (JN5139/JN5121 Version)

```
bool_t bAHI_IpSendData(uint8 u8Length,
                       uint8 *pau8Data);
```

Description

This function is used on the JN5139/JN5121 device to copy data from RAM to the IP Transmit buffer and to indicate that data is ready to be transmitted across the IP interface to the remote processor (the SPI master).

The function requires the data length to be specified, as well as a pointer to a RAM buffer containing the data. The data should be stored in the RAM buffer according to the byte order (Big or Little Endian) specified in the function **vAHI_IpEnable()**.

The function copies the specified data to the IP Transmit buffer, ready to be sent when the master device initiates the transfer. The IP_INT pin is also asserted to indicate to the master that data is ready to be sent.

The data length is transmitted in the first 32-bit word of the data payload. It is the responsibility of the SPI master receiving the data to retrieve the data length from the payload.

Parameters

u8Length	Length of data to be sent (in 32-bit words)
**pau8Data*	Pointer to RAM buffer containing the data to be sent

Returns

TRUE if successful, FALSE if unable to send

bAHI_IpReadData (JN5148 Version)

```
bool_t bAHI_IpReadData(uint8 *pu8Length,
                       uint8 *pau8Data,
                       bool_t bEndian);
```

Description

This function is used on the JN5148 device to copy received data from the IP Receive buffer into RAM.

The function must provide a pointer to a RAM buffer to receive the data and a pointer to a RAM location to receive the data length.

Data is stored in the specified RAM buffer according to the specified byte order (Big or Little Endian).

After the data has been read, the function **vAHI_IpReadyToReceive()** can be used to indicate to the SPI master that the IP interface is ready to receive more data.

Parameters

**pu8Length*	Pointer to location to receive data length (in 32-bit words)
**pau8Data*	Pointer to RAM buffer to receive data
bEndian	Byte order (Big or Little Endian) for storing data: E_AHI_IP_BIG_ENDIAN E_AHI_IP_LITTLE_ENDIAN

Returns

TRUE if data read successfully, FALSE if unable to raed

bAHI_IpReadData (JN5139/JN5121 Version)

```
bool_t bAHI_IpReadData(uint8 *pu8Length,
                       uint8 *pau8Data);
```

Description

This function is used on the JN5139/JN5121 device to copy received data from the IP Receive buffer into RAM.

The function must provide a pointer to a RAM buffer to receive the data and a pointer to a RAM location to receive the data length.

Data is stored in the specified RAM buffer according to the specified byte order (Big or Little Endian) specified in the function **vAHI_IpEnable()**.

After the data has been read, the IP interface will indicate to the SPI master that the interface is ready to receive more data.

Parameters

**pu8Length*	Pointer to location to receive data length (in 32-bit words)
**pau8Data*	Pointer to RAM buffer to receive data

Returns

TRUE if data read successfully, FALSE if unable to read

bAHI_IpTxDone

```
bool_t bAHI_IpTxDone (void);
```

Description

This function checks whether data copied to the IP Transmit buffer has been sent to the remote processor (the SPI master).

Parameters

None

Returns

TRUE if data sent, FALSE if incomplete

bAHI_IpRxDataAvailable

```
PUBLIC bool_t bAHI_IpRxDataAvailable(void);
```

Description

This function checks whether data from the remote processor (the SPI master) has been received in the IP Receive buffer.

Parameters

None

Returns

TRUE if IP Receive buffer contains data, FALSE otherwise

vAHI_IpReadyToReceive (JN5148 Only)

```
void vAHI_IpReadyToReceive(void);
```

Description

This function is used to indicate that the IP Receive buffer is free to receive data from the remote processor (the SPI master).

Parameters

None

Returns

None

vAHI_IpRegisterCallback

void vAHI_IpRegisterCallback(
PR_HWINT_APPCALLBACK *prIpCallback***);**

Description

This function registers an application callback that will be called when the SPI interrupt is triggered. The interrupt is generated when either a transmit or receive transaction has completed.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A.

Parameters

prIpCallback	Pointer to callback function to be registered

Returns

None

15. Digital Audio Interface (JN5148 Only)

This chapter details the functions for controlling the 4-wire Digital Audio Interface (DAI) on the Jennic JN5148 wireless microcontroller. This interface allows communication with external devices that support various digital audio interfaces such as CODECs.

Audio data can be input to the DAI on DIO13 and/or output from the DAI on DIO18. The bit clock from the DAI is output on DIO17, and DIO12 is used for Word Select (WS).

The data path between the CPU and the DAI can optionally be buffered using the Sample FIFO Interface - the functions for configuring and monitoring this interface are detailed in Chapter 16.

The DAI functions are listed below, along with their page references:

Function	Page
vAHI_DaiEnable (JN5148 Only)	246
vAHI_DaiSetBitClock (JN5148 Only)	247
vAHI_DaiSetAudioData (JN5148 Only)	248
vAHI_DaiSetAudioFormat (JN5148 Only)	249
vAHI_DaiConnectToFIFO (JN5148 Only)	250
vAHI_DaiWriteAudioData (JN5148 Only)	251
vAHI_DaiReadAudioData (JN5148 Only)	252
vAHI_DaiStartTransaction (JN5148 Only)	253
bAHI_DaiPollBusy (JN5148 Only)	254
vAHI_DaiInterruptEnable (JN5148 Only)	255
vAHI_DaiRegisterCallback (JN5148 Only)	256

Note: For guidance on using the DAI functions in JN5148 application code, refer to Chapter 15 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

vAHI_DaiEnable (JN5148 Only)

void vAHI_DaiEnable(bool_t *bEnable***);**

Description

This function can be used to enable or disable the Digital Audio Interface (DAI) - that is, to power up or power down the interface.

Parameters

bEnable	Enable/disable the DAI: TRUE - enable (power up) FALSE - disable (power down)

Returns

None

vAHI_DaiSetBitClock (JN5148 Only)

```
void vAHI_DaiSetBitClock(uint8 u8Div, bool_t bConClock);
```

Description

This function can be used to configure the DAI bit clock, derived from the 16-MHz system clock.

The 16-MHz system clock is divided by twice the specified division factor to produce the bit clock. Division factors can be specified in the range 0 to 63, allowing division by up to 126. If a zero division factor is specified, the divisor used will be 2. Thus, the maximum possible bit clock frequency is 8 MHz. The default division factor is 8, giving a divisor of 16 and a bit clock frequency of 1 MHz.

The bit clock is output on DIO17 to synchronise data between the (master) interface and an external CODEC. It can be output either permanently or only during data transfers.

Parameters

u8Div	Division factor, in the range 0 to 63 - the 16-MHz system clock will be divided by 2 x *u8Div*, or 2 if *u8Div*=0
bConClock	Bit clock output enable: TRUE - enable clock output permanently FALSE - enable clock output only during data transfers

Returns

None

vAHI_DaiSetAudioData (JN5148 Only)

void vAHI_DaiSetAudioData(uint8 *u8CharLen*,
bool_t *bPadDis*,
bool_t *bExPadEn*,
uint8 *u8ExPadLen***);**

Description

This function configures the size and padding options of a data transfer between the DAI and an external audio device. These values should be set to match the requirements of the external device.

The number of data bits in the transfer can be specified in the range 1 to 16 per stereo channel. The function also allows padding bits (zeros) to be inserted after the data bits to make the data transfer up to a certain size:

- Padding can be enabled/disabled using the parameter *bPadDis*.
- The default padding automatically makes the transfer size up to 16 bits per channel. Extra padding bits can be added to increase the transfer size per channel to a value between 17 and 32 bits. This option is enabled using the parameter *bExPadEn* (padding must also be enabled through *bPadDis*).
- If extra padding is enabled (through *bExPadEn*), the number of additional padding bits needed to achieve the required transfer size is specified through *u8ExPadLen*. Note that padding bits will be automatically added to reach 16 bits and the extra padding bits are those required to increase the transfer size from 16 bits (e.g. add 8 extra padding bits to achieve a 24-bit transfer size). This option allows data transfer sizes of up to 32 bits per channel (16 data bits and 16 padding bits).

Parameters

u8CharLen	Number of data bits per stereo channel: 0: 1 bit 1: 2 bits : 15: 16 bits
bPadDis	Disable/enable automatic data padding: TRUE - disable padding FALSE - enable padding
bExPadEn	Enable/disable extra data padding for transfer sizes greater than 16 bits (extra padding bits specified via *u8ExPadLen*): TRUE - enable extra padding FALSE - disable extra padding
u8ExPadLen	Number of extra padding bits to increase transfer size from 16 bits to desired size (only valid if *bExPadEn* set to TRUE): 0: 1 bit 1: 2 bits : 15: 16 bits

Returns

None

vAHI_DaiSetAudioFormat (JN5148 Only)

```
void vAHI_DaiSetAudioFormat(uint8 u8Mode,
                            bool_t bWsIdle,
                            bool_t bWsPolarity);
```

Description

This function is used to configure the audio data format to one of:

- Left-justified mode
- Right-justified mode
- I^{2}S-compatible mode

The function also allows the word-select (WS) signal to be configured - this signal indicates which stereo channel is being transmitted. Normally, it is asserted (1) for the right channel and de-asserted (0) for the left channel, as in I^{2}S.

Parameters

u8Mode	Transfer mode: 00: I^{2}S-compatible (left-justified, MSB 1 cycle after WS) 01: Left-justified (MSB coincident with assertion of WS) 1x: Right-justified (LSB coincident with de-assertion of WS)
bWsIdle	WS setting during idle time: TRUE - Left channel (so there is always a transition at the end of the transfer). May be used for right-justified transfer mode FALSE - Right channel (so there is always a transition at the start of the transfer).Should be used for left-justified and I^{2}S-compatible transfer modes
bWsPolarity	WS polarity: 1: WS inverted 0: WS not inverted (as in I^{2}S)

Returns

None

vAHI_DaiConnectToFIFO (JN5148 Only)

```
void vAHI_DaiConnectToFIFO(bool_t bMode,
                           bool_t bChannel);
```

Description

This function can be used to connect the DAI to the Sample FIFO auxiliary interface, which can be used to store a mono audio sample corresponding to one of the stereo audio channels of the DAI - the left channel or right channel can be selected.

Timer 2 is configured to provide the timing source for samples transferred via the DAI. A rising edge on the PWM line of Timer 2 causes a single DAI transfer, with data transferred to/from the Sample FIFO.

Parameters

u8Mode	Enable/disable Sample FIFO auxiliary mode: TRUE - enable (DAI controlled by Sample FIFO and Timer 2) FALSE - disable
bChannel	Channel to contain data corresponding to mono sample: TRUE - right channel FALSE - left channel

Returns

None

vAHI_DaiWriteAudioData (JN5148 Only)

```
void vAHI_DaiWriteAudioData(uint16 u16TxDataR,
                            uint16 u16TxDataL);
```

Description

This function writes audio data into the DAI Transmit buffer, ready for transmission to an external audio device. The left- and right-channel data are specified separately.

The written data can be subsequently transmitted by calling the function **vAHI_DaiStartTransaction()**.

Note that this write function cannot be used if the auxiliary Sample FIFO interface is enabled.

Parameters

u16TxDataR	Right-channel data to transmit
u16TxDataL	Left-channel data to transmit

Returns

None

vAHI_DaiReadAudioData (JN5148 Only)

```
void vAHI_DaiReadAudioData(uint16 *pu16RxDataR,
                           uint16 *pu16RxDataL);
```

Description

This function reads audio data received in the DAI Receive buffer from an external audio device. The left and right channels are extracted separately. This function should be called following a successful poll using **bAHI_DaiPollBusy()** or, if interrupts are enabled, in the user-defined callback function registered using **vAHI_DaiRegisterCallback()**.

Note that this read function cannot be used if the auxiliary Sample FIFO interface is enabled.

Parameters

pu16RxDataR	Pointer to location where right-channel data will be placed
pu16RxDataL	Pointer to location where left-channel data will be placed

Returns

None

vAHI_DaiStartTransaction (JN5148 Only)

```
void vAHI_DaiStartTransaction(void);
```

Description

This function starts a DAI transaction - that is, a data transfer to/from the attached external audio device. After calling this function, data is transmitted from the DAI Transmit buffer to the external device and data from the external device is received in the DAI Receive buffer.

Note that this function cannot be used when operating the DAI in conjunction with the auxiliary Sample FIFO interface.

Parameters

None

Returns

None

bAHI_DaiPollBusy (JN5148 Only)

```
bool_t bAHI_DaiPollBusy(void);
```

Description

This function can be used to determine whether the DAI is busy performing a data transfer (including cases where the auxiliary Sample FIFO interface is being used to control the transfer).

Parameters

None

Returns

Status of the DAI:

TRUE - busy
FALSE - not busy

vAHI_DaiInterruptEnable (JN5148 Only)

```
void vAHI_DaiInterruptEnable(bool_t bEnable);
```

Description

This function can be used to enable/disable DAI interrupts.

If interrupts are enabled, an interrupt will be generated at the end of each data transfer via the DAI. If interrupts are disabled, an alternative way of determining whether a data transfer via the DAI has completed is to call the function **bAHI_DaiPollBusy()**.

Parameters

bEnable	Enable/disable DAI interrupts: TRUE - enable FALSE - disable

Returns

None

vAHI_DaiRegisterCallback (JN5148 Only)

```
void vAHI_DaiRegisterCallback(
                 PR_HWINT_APPCALLBACK prDaiCallback);
```

Description

This function registers a user-defined callback function that will be called when the DAI interrupt is triggered.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A.

Parameters

prDaiCallback	Pointer to callback function to be registered

Returns

None

16. Sample FIFO Interface (JN5148 Only)

This chapter details the functions for controlling and monitoring the Sample FIFO Interface of the Jennic JN5148 wireless microcontroller. This interface is a 10-deep FIFO that can be implemented between the CPU and the DAI (Digital Audio Interface). The FIFO can handle data transfers in either direction (CPU to DAI or DAI to CPU).

Each of the ten samples in the FIFO is a 16-bit value. Therefore, used in conjunction with the DAI, the FIFO can only handle 16-bit mono audio data (taken from either the left or right channel of the full stereo audio sample handled by the DAI). Also, Timer 2 is used to generate an internal timing signal to control the flow of data to/from the DAI.

The Sample FIFO Interface can reduce the rate at which the CPU needs to generate output data and/or process input data, which may allow more efficient operation.

The Sample FIFO functions are listed below, along with their page references:

Function	**Page**
vAHI_FifoEnable (JN5148 Only)	258
bAHI_FifoRead (JN5148 Only)	259
vAHI_FifoWrite (JN5148 Only)	260
u8AHI_FifoReadRxLevel (JN5148 Only)	261
u8AHI_FifoReadTxLevel (JN5148 Only)	262
vAHI_FifoSetInterruptLevel (JN5148 Only)	263
vAHI_FifoEnableInterrupts (JN5148 Only)	264
vAHI_FifoRegisterCallback (JN5148 Only)	265

Note: For guidance on using the Sample FIFO functions in JN5148 application code, refer to Chapter 16 of the *Integrated Peripherals API User Guide (JN-UG-3066)*.

vAHI_FifoEnable (JN5148 Only)

```
void vAHI_FifoEnable(bool_t bEnable);
```

Description

This function can be used to enable or disable the Sample FIFO Interface.

Parameters

bEnable	Enable/disable the Sample FIFO Interface: TRUE - enable FALSE - disable

Returns

None

bAHI_FifoRead (JN5148 Only)

bool_t bAHI_FifoRead(uint16 ****pu16RxData*);**

Description

This function can be used to read the next available received data sample from the Sample FIFO.

Parameters

pu16RxData	Pointer to the location to receive the read value

Returns

TRUE: Read value is valid

FALSE: Reda value is invalid

vAHI_FifoWrite (JN5148 Only)

```
void vAHI_FifoWrite(uint16 u16TxBuffer);
```

Description

This function can be used to write a data value to the Sample FIFO for transmission.

Parameters

u16TxBuffer	16-bit data value to be written to the FIFO

Returns

None

u8AHI_FifoReadRxLevel (JN5148 Only)

```
uint8 u8AHI_FifoReadRxLevel(void);
```

Description

This function can be used to obtain the Receive level of the Sample FIFO.

Parameters

None

Returns

FIFO Receive level obtained

u8AHI_FifoReadTxLevel (JN5148 Only)

```
uint8 u8AHI_FifoReadTxLevel(void);
```

Description

This function can be used to obtain the Transmit level of the Sample FIFO.

Parameters

None

Returns

FIFO Transmit level obtained

vAHI_FifoSetInterruptLevel (JN5148 Only)

```
void vAHI_FifoSetInterruptLevel(uint8 u8RxIntLevel,
                                uint8 u8TxIntLevel,
                                bool_t bDataSource);
```

Description

This function can be used to set the Receive and Transmit interrupt levels for the Sample FIFO:

- The fill-level of the FIFO above which a Receive interrupt will be triggered (to signal that the FIFO should be read)
- The fill-level of the FIFO below which a Transmit interrupt will be triggered (to signal that the FIFO should be re-filled)

Sample FIFO interrupts are enabled using **vAHI_FifoEnableInterrupts()**.

Parameters

u8RxIntLevel	FIFO fill-level above which a Receive interrupt will occur
u8TxIntLevel	FIFO fill-level below which a Transmit interrupt will occur
bDataSource	Peripheral with which Sample FIFO Interface exchanges data: TRUE - connect to DAI FALSE - reserved (do not use)

Returns

None

vAHI_FifoEnableInterrupts (JN5148 Only)

```
void vAHI_FifoEnableInterrupts(bool_t bRxAbove,
                               bool_t bTxBelow,
                               bool_t bRxOverflow,
                               bool_t bTxEmpty);
```

Description

This function can be used to individually enable/disable the four types of Sample FIFO interrupt:

- **Receive Interrupt:** This is generated when the FIFO fill-level rises above a threshold pre-defined using **vAHI_FifoSetInterruptLevel()**. This interrupt can be used to prompt a read of the FIFO to collect received data.
- **Transmit Interrupt:** This is generated when the FIFO fill-level falls below a threshold pre-defined using **vAHI_FifoSetInterruptLevel()**. This interrupt can be used to prompt a write to the FIFO to provide further data to be transmitted.
- **Receive Overflow Interrupt:** This is generated when the FIFO has been filled to its maximum capacity and an attempt has been made to add more received data to the FIFO. This interrupt can be used to prompt a read of the FIFO to collect received data.
- **Transmit Empty Interrupt:** This is generated when the FIFO becomes empty and there is no more data to be transmitted. This interrupt can be used to prompt a write to the FIFO to provide further data to be transmitted.

Parameters

bRxAbove	Enable/disable Receive interrupts: TRUE - enable FALSE - disable
bTxBelow	Enable/disable Transmit interrupts: TRUE - enable FALSE - disable
bRxOverflow	Enable/disable Receive Overflow interrupts: TRUE - enable FALSE - disable
bTxEmpty	Enable/disable Transmit Empty interrupts: TRUE - enable FALSE - disable

Returns

None

vAHI_FifoRegisterCallback (JN5148 Only)

```
void vAHI_FifoRegisterCallback(
                    PR_HWINT_APPCALLBACK prFifoCallback);
```

Description

This function registers a user-defined callback function that will be called when the Sample FIFO Interface interrupt is triggered.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.

Interrupt handling is described in Appendix A.

Parameters

prFifoCallback	Pointer to callback function to be registered

Returns

None

17. External Flash Memory

This chapter describes functions for erasing and programming a sector of an external Flash memory device. Jennic modules are supplied with Flash memory devices fitted, but the functions can also be used with custom modules which have different Flash devices.

For some operations, two versions of the relevant function are provided, as follows:

- A function designed to interact with a 128-KB Flash device in which the application data is stored in the final sector (Sector 3), e.g. the ST M25P10 Flash device fitted to JN5121 modules - these functions are designed to access Sector 3 only and all addresses are offsets from the start of Sector 3.
- A function designed to interact with a 128-KB or 512-KB Flash device, and which is able to access any sector - it is usual to store application data in the final sector, detailed in the table below for the different Flash devices:

Flash Device	**Final Sector**		
	Number	**Start Address**	**Length (KB)**
AT25F512	2	0x8000	32
SST25V010	3	0x18000	32
M25P10A	3	0x18000	32
M25P40	7	0x70000	64

Table 1: Final Sectors of Different Flash Devices

To access sectors other than the final sector, you should refer to the data sheet for the Flash device to obtain the necessary sector details. However, be careful not to erase essential data such as application code. The application is stored from the start of the Flash memory. It is therefore normally held in Sectors 0, 1 and 2 of a 128-KB device, and in Sectors 0 and 1 of a 512-KB device.

The Flash memory functions are listed below, along with their page references:

Function	**Page**
bAHI_FlashInit	268
bAHI_FlashErase (JN5121/JN5139 Only)	269
bAHI_FlashEraseSector	270
bAHI_FlashProgram (JN5121/JN5139 Only)	271
bAHI_FullFlashProgram	272
bAHI_FlashRead (JN5121/JN5139 Only)	272
bAHI_FullFlashRead	274
vAHI_FlashPowerDown (JN5139/JN5148 Only)	275
vAHI_FlashPowerUp (JN5139/JN5148 Only)	276

bAHI_FlashInit

```
bool_t bAHI_FlashInit(
                teFlashChipType flashType,
                tSPIflashFncTable *pCustomFncTable);
```

Description

This function selects the type of external Flash memory device to be used.

The Flash memory device can be one of four supported device types or a custom device. In the latter case, a custom table of functions must be supplied for interaction with the device. Auto-detection of the Flash device type can also be selected.

Note that this function can be called on the JN5121 wireless microcontroller but has no effect and always returns TRUE (JN5121 only supports the ST M25P10A).

Parameters

flashType	Type of Flash memory device, one of: E_FL_CHIP_ST_M25P10_A (ST M25P10A) E_FL_CHIP_ST_M25P40_A (ST M25P40) E_FL_CHIP_SST_25VF010 (SST 25VF010) E_FL_CHIP_ATMEL_AT25F512 (Atmel AT25F512) E_FL_CHIP_CUSTOM (custom device) E_FL_CHIP_AUTO (auto-detection)
**pCustomFncTable*	Pointer to the custom function table for a custom Flash device (E_FL_CHIP_CUSTOM). If a supported Flash device is used, set to NULL.

Returns

TRUE if initialisation was successful, FALSE if failed

bAHI_FlashErase (JN5121/JN5139 Only)

```
bool_t bAHI_FlashErase(void);
```

Description

This function erases the 32-KB sector of Flash memory (JN5121/JN5139 only) used to store application data, by setting all bits to 1. The function does not affect sectors containing application code.

Caution: *This function can only be used with 128-KB Flash memory devices with four 32-KB sectors (numbered 0 to 3), where application data is stored in Sector 3.*

Parameters

None

Returns

TRUE if sector erase was successful, FALSE if erase failed

bAHI_FlashEraseSector

bool_t bAHI_FlashEraseSector(uint8 *u8Sector***);**

Description

This function erases the specified sector of Flash memory by setting all bits to 1.

The function can be used with 128-KB and 512-KB Flash memory devices with up to 8 sectors. Refer to the datasheet of the Flash memory device for details of its sectors.

Caution: *Be careful not to erase essential data such as application code. The application is stored from the start of the Flash memory. It is therefore normally held in Sectors 0, 1 and 2 of a 128-KB device, and in Sectors 0 and 1 of a 512-KB device.*

Parameters

u8Sector	Number of the sector to be erased (in the range 2 to 7)

Returns

TRUE if sector erase was successful, FALSE if erase failed

bAHI_FlashProgram (JN5121/JN5139 Only)

bool_t bAHI_FlashProgram(uint16 *u16Addr*,
uint16 *u16Len*,
uint8 **pu8Data*);

Description

This function programs a block of Flash memory (JN5121/JN5139 only) by clearing the appropriate bits from 1 to 0.

This mechanism does not allow bits to be set from 0 to 1. It is only possible to set bits to 1 by erasing the entire sector - therefore, before using this function, you must call the function **bAHI_FlashErase()**.

***Caution:** This function can only be used with 128-KB Flash memory devices with four 32-KB sectors (numbered 0 to 3), where application data is stored in Sector 3. Consequently, the start address specified in this function is an offset within this area, i.e. it starts at 0.*

Parameters

u16Addr	Address offset of first Flash memory byte to be programmed (offset from start of 32-KB block)
u16Len	Number of bytes to be programmed (integer in the range 1 to 0x8000)
**pu8Data*	Pointer to start of data block to be written to Flash memory

Returns

TRUE if write was successful

FALSE if write failed or input parameters were invalid

bAHI_FullFlashProgram

```
bool_t bAHI_FullFlashProgram(uint32 u32Addr,
                             uint16 u16Len,
                             uint8 *pu8Data);
```

Description

This function programs a block of Flash memory by clearing the appropriate bits from 1 to 0. The function can be used to access any sector of a 128-KB or 512-KB Flash memory device.

This mechanism does not allow bits to be set from 0 to 1. It is only possible to set bits to 1 by erasing the entire sector - therefore, before using this function, you must call the function **bAHI_FlashEraseSector()**.

Parameters

u32Addr	Address of first Flash memory byte to be programmed
u16Len	Number of bytes to be programmed (integer in the range 1 to 0x8000)
**pu8Data*	Pointer to start of data block to be written to Flash memory

Returns

TRUE if write was successful

FALSE if write failed

bAHI_FlashRead (JN5121/JN5139 Only)

```
bool_t bAHI_FlashRead(uint16 u16Addr,
                      uint16 u16Len,
                      uint8 *pu8Data);
```

Description

This function reads data from the application data area of Flash memory (JN5121/ JN5139 only).

***Caution:** This function can only be used with 128-KB Flash memory devices with four 32-KB sectors (numbered 0 to 3), where application data is stored in Sector 3. Consequently, the start address specified in this function is an offset within this area, i.e. it starts at 0.*

If the function parameters are invalid (e.g. by trying to read beyond end of sector), the function returns without reading anything.

Parameters

u16Addr	Address offset of first Flash memory byte to be read (offset from start of 32-KB block)
u16Len	Number of bytes to be read (integer in the range 1 to 0x8000)
**pu8Data*	Pointer to start of buffer to receive read data

Returns

TRUE if read was successful

FALSE if read failed or input parameters were invalid

bAHI_FullFlashRead

```
bool_t bAHI_FullFlashRead(uint32 u32Addr,
                          uint16 u16Len,
                          uint8 *pu8Data);
```

Description

This function reads data from the application data area of Flash memory. The function can be used to access any sector of a 128-KB or 512-KB Flash memory device.

If the function parameters are invalid (e.g. by trying to read beyond end of sector), the function returns without reading anything.

Parameters

u32Addr	Address of first Flash memory byte to be read
u16Len	Number of bytes to be read: integer in range 1 to 0x8000
**pu8Data*	Pointer to start of buffer to receive read data.

Returns

TRUE (always)

vAHI_FlashPowerDown (JN5139/JN5148 Only)

```
void vAHI_FlashPowerDown(void);
```

Description

This function sends a ‘power down’ command to the Flash memory device attached to the JN5139/JN5148 wireless microcontroller. This allows further power savings to be made when the wireless microcontroller is put into a sleep mode (other than deep sleep mode, for which the Flash memory device is powered down automatically).

The following Flash devices are supported by this function:

- STM25P10A - for JN5139 and JN5148 devices
- STM25P40 - for JN5148 device only

If the function is called for an unsupported Flash device, the function will return without doing anything.

If the Flash device is to be unpowered while the JN5139/JN5148 device is sleeping, this function must be called before **vAHI_Sleep()** is called to put the CPU into sleep mode. However, note that in the case of deep sleep mode, the Flash device is automatically powered down before the JN5139/JN5148 enters deep sleep mode and therefore there is no need to call **vAHI_FlashPowerDown()**.

If you use this function before entering ‘sleep without memory held’ then the boot loader will automatically power up the Flash memory device during the wake-up sequence. However, if you use the function before entering ‘sleep with memory held’ then the boot loader will not power up Flash memory on waking. In the latter case, you must power up the device using **vAHI_FlashPowerUp()** after waking and before attempting to access the Flash memory.

Parameters

None

Returns

None

vAHI_FlashPowerUp (JN5139/JN5148 Only)

```
void vAHI_FlashPowerUp(void);
```

Description

This function sends a ‘power up’ command to the Flash memory device attached to the JN5139/JN5148 wireless microcontroller.

The following Flash devices are supported by this function:

- STM25P10A - for JN5139 and JN5148 devices
- STM25P40 - for JN5148 device only

If the function is called for an unsupported Flash device, the function will return without doing anything.

This function must be called when the JN5139/JN5148 device wakes from ‘sleep without memory held’ if the Flash device was powered down using **vAHI_FlashPowerDown()** before the JN5139/JN5148 device entered sleep mode.

However, note that in the case of ‘sleep with memory held’ and deep sleep mode, the Flash device is automatically powered up when the JN5139/JN5148 wakes from sleep and therefore there is no need to call **vAHI_FlashPowerUp()**.

Parameters

None

Returns

None

Appendices

A. Interrupt Handling

Interrupts from the on-chip peripherals are handled by a set of peripheral-specific callback functions. These user-defined functions can be introduced using the appropriate callback registration functions of the Integrated Peripherals API. For example, you can write your own interrupt handler for UART0 and then register this callback function using the **vAHI_Uart0RegisterCallback()** function.

Note: A callback function is executed in interrupt context. You must therefore ensure that the function returns to the main program in a timely manner.

***Caution:** Registered callback functions are only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, any callback functions must be re-registered before calling **u32AHI_Init()** on waking.*

The callback function prototype and related interrupt resources are detailed in the sub-sections below. For further information on interrupt handling, refer to the *Integrated Peripherals API User Guide (JN-UG-3066)*.

A.1 Callback Function Prototype and Parameters

All peripheral-specific callback functions must have the following prototype:

```
void vHwDeviceIntCallback(uint32 u32DeviceId,
                          uint32 u32ItemBitmap);
```

The parameters of this function prototype are as follows:

- *u32DeviceId* identifies the peripheral that generated the interrupt. Enumerations for the possible interrupt sources are provided in the API and are detailed in Appendix A.2.
- *u32ItemBitmap* is a bitmap that identifies the specific cause of the interrupt within the peripheral block identified through *u32DeviceId* above. Masks are provided in the API that allow particular interrupt causes to be checked for. The UART interrupts are an exception as, in their case, an enumerated value is passed via this parameter instead of a bitmap. The masks and enumerations are detailed in Appendix A.3.

A.2 Peripheral Interrupt Enumerations (u32DeviceId)

The device ID, *u32DeviceId*, is an enumerated value indicating the peripheral that generated the interrupt. The enumerations are detailed in Table 1 below.

Enumeration	Interrupt Source	Callback Registration Function
E_AHI_DEVICE_SYSCTRL	System Controller	**vAHI_SysCtrlRegisterCallback()**
E_AHI_DEVICE_ANALOGUE	Analogue Peripherals	**vAHI_APRegisterCallback()**
E_AHI_DEVICE_UART0	UART 0	**vAHI_Uart0RegisterCallback()**
E_AHI_DEVICE_UART1	UART 1	**vAHI_Uart1RegisterCallback()**
E_AHI_DEVICE_TIMER0	Timer 0	**vAHI_Timer0RegisterCallback()**
E_AHI_DEVICE_TIMER1	Timer 1	**vAHI_Timer1RegisterCallback()**
E_AHI_DEVICE_TIMER2	Timer 2 *	**vAHI_Timer2RegisterCallback()**
E_AHI_DEVICE_TICK_TIMER	Tick Timer	**vAHI_TickTimerRegisterCallback() *** **vAHI_TickTimerInit()**
E_AHI_DEVICE_SI **	Serial Interface (2-wire)	**vAHI_SiRegisterCallback() ****
E_AHI_DEVICE_SPIM	SPI Master	**vAHI_SpiRegisterCallback()**
E_AHI_DEVICE_INTPER	Intelligent Peripheral	**vAHI_IpRegisterCallback()**
E_AHI_DEVICE_I2S	Digital Audio Interface *	**vAHI_DaiRegisterCallback()**
E_AHI_DEVICE_AUDIOFIFO	Sample FIFO Interface *	**vAHI_FifoRegisterCallback()**
E_AHI_DEVICE_AES	Encryption engine	Refer to *AES Coprocessor API Reference Manual (JN-RM-2013)*

Table 1: u32DeviceId Enumerations

* JN5148 device only

** Used for both SI master and SI slave interrupts

A.3 Peripheral Interrupt Sources (u32ItemBitmap)

The parameter *u32ItemBitmap* is a 32-bit bitmask indicating the individual interrupt source within the peripheral (except for the UARTs, for which the parameter returns an enumerated value). The bits and their meanings are detailed in the tables below.

Mask (Bit)	Description
E_AHI_SYSCTRL_CKEM_MASK (31) *	System clock source has been changed
E_AHI_SYSCTRL_RNDEM_MASK (30)	A new value has been generated by the Random Number Generator (JN5148 only)
E_AHI_SYSCTRL_COMP1_MASK (29) E_AHI_SYSCTRL_COMP0_MASK (28)	Comparator (0 and 1) events
E_AHI_SYSCTRL_WK1_MASK (27) E_AHI_SYSCTRL_WK0_MASK (26)	Wake Timer events
E_AHI_SYSCTRL_VREM_MASK (25) * E_AHI_SYSCTRL_VFEM_MASK (24) *	Brownout condition entered Brownout condition exited
E_AHI_SYSCTRL_PC1_MASK (23) E_AHI_SYSCTRL_PC0_MASK (22)	Pulse Counter (0 or 1) has reached its pre-configured reference value (JN5148 only)
E_AHI_DIO20_INT (20) E_AHI_DIO19_INT (19) E_AHI_DIO18_INT (18) . . . E_AHI_DIO0_INT (0)	Digital IO (DIO) events

Table 2: System Controller

* JN5148 device only

Mask (Bit)	Description
E_AHI_AP_ACC_INT_STATUS_MASK (1 and 0)	Asserted in ADC accumulation mode to indicate that conversion is complete and the accumulated sample is available
E_AHI_AP_CAPT_INT_STATUS_MASK (0)	Asserted in all ADC modes to indicate that an individual conversion is complete and the resulting sample is available

Table 3: Analogue Peripherals

Mask (Bit)	Description
E_AHI_TIMER_RISE_MASK (1)	Interrupt status, generated on timer rising edge (low-to-high transition) - will be non-zero if interrupt for timer rising output has been set
E_AHI_TIMER_PERIOD_MASK (0)	Interrupt status, generated on end of timer period (high-to-low transition) - will be non-zero if interrupt for end of timer period has been set

Table 4: Timers (identical for all timers)

Mask (Bit)	Description
0	Single source for Tick-timer interrupt, therefore returns 1 every time

Table 5: Tick Timer

Mask (Bit)	Description
E_AHI_SIM_RXACK_MASK (7)	Asserted if no acknowledgement is received from the addressed slave
E_AHI_SIM_BUSY_MASK (6)	Asserted if a START signal is detected Cleared if a STOP signal is detected
E_AHI_SIM_AL_MASK (5)	Asserted to indicate loss of arbitration
E_AHI_SIM_ICMD_MASK (2)	Asserted to indicate invalid command
E_AHI_SIM_TIP_MASK (1)	Asserted to indicate transfer in progress
E_AHI_SIM_INT_STATUS_MASK (0)	Interrupt status - interrupt indicates loss of arbitration or that byte transfer has completed

Table 6: Serial Interface (2-wire) Master

Mask (Bit)	Description
E_AHI_SIS_ERROR_MASK (4)	I^{2}C protocol error
E_AHI_SIS_LAST_DATA_MASK (3)	Last data transferred (end of burst)
E_AHI_SIS_DATA_WA_MASK (2)	Buffer contains data to be read by SI slave
E_AHI_SIS_DATA_RTKN_MASK (1)	Data taken from buffer by SI master (buffer free for next data to be loaded)
E_AHI_SIS_DATA_RR_MASK (0)	Buffer needs loading with data for SI master

Table 7: Serial Interface (2-wire) Slave

Mask (Bit)	Description
E_AHI_SPIM_TX_MASK (0)	Transfer has completed

Table 8: SPI Master

Mask (Bit)	Description
E_AHI_DAI_INT_MASK (0)	End of data transfer via the DAI

Table 9: Digitial Audio Interface

Mask (Bit)	Description
E_AHI_INT_RX_FIFO_HIGH_MASK (3)	Rx FIFO is nearly full and needs to be read
E_AHI_INT_TX_FIFO_LOW_MASK (2)	Tx FIFO is nearly empty and needs more data
E_AHI_INT_RX_FIFO_OVERFLOW_MASK (1)	Rx FIFO is full/overflowing and must be read
E_AHI_INT_TX_FIFO_EMPTY_MASK (0)	Tx FIFO is empty and needs data

Table 10: Sample FIFO Interface

Mask (Bit)	Description
E_AHI_IP_INT_STATUS_MASK (6)	Transaction has completed, i.e. slave-select goes high and TXGO or RXGO has gone low
E_AHI_IP_TXGO_MASK (1)	Asserted when transmit data is copied to the internal buffer and cleared when it has been transmitted
E_AHI_IP_RXGO_MASK (0)	Asserted when device is in ready-to-receive state and cleared when data reception is complete

Table 11: Intelligent Peripheral

For the UART interrupts, *u32ItemBitmap* returns the following enumerated values:

Enumerated Value	Description (and Priority)
E_AHI_UART_INT_RXLINE (3)	Receive line status (highest priority)
E_AHI_UART_INT_RXDATA (2)	Receive data available (next highest priority)
E_AHI_UART_INT_TIMEOUT (6)	Time-out indication (next highest priority)
E_AHI_UART_INT_TX (1)	Transmit FIFO empty (next highest priority)
E_AHI_UART_INT_MODEM (0)	Modem status (lowest priority)

Table 12: UART (identical for both UARTs)

Table 12 lists the UART interrupts from highest priority to lowest priority.

Revision History

Version	**Date**	**Comments**
1.0-1.10	2005-2006	Editions for JN5121 device
2.0-2.8	2007-2009	Editions for JN5121 and JN5139 devices
3.0	29-Sep-2009	Support for JN5148 device added
3.1	21-Dec-2009	Various updates/corrections made and Timers chapter re-worked
3.2	16-July-2010	Re-worked to accompany first release of *Integrated Peripherals API User Guide (JN-UG-3066)*

Important Notice

Jennic Ltd
Furnival Street
Sheffield
S1 4QT
United Kingdom

Tel: +44 (0)114 281 2655
Fax: +44 (0)114 281 2951
E-mail: info@jennic.com

For the contact details of your local Jennic office or distributor, refer to the Jennic web site:

www.Jennic.com
TECHNOLOGY FOR A CHANGING WORLD