JN51xx Integrated Peripherals API User Guide

JN-UG-3066
Revision 2.0
24 November 2010

Contents

Part III: Appendices

About this Manual

This manual describes the use of the JN51xx Integrated Peripherals Application Programming Interface (API) to interact with the peripherals on the NXP JN5148 and JN5139 microcontrollers. The manual explains the basic operation of each peripheral and indicates how to use the relevant API functions to control the peripheral from the application which runs on the JN51xx device. The C functions and associated resources of the API are fully detailed.

Note 1: Not all of the peripherals described in this manual are featured on both the JN5148 and JN5139 devices. Where a peripheral is restricted to one of these devices, this will be indicated.

Note 2: This manual incorporates information from the former *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

Organisation

This manual is divided into three parts:

- Part I: Concept and Operational Information comprises 17 chapters:
  - Chapter 1 presents a functional overview of the JN51xx Integrated Peripherals API.
  - Chapter 2 describes use of the **General functions** of the API, including the API initialisation function.
  - Chapter 3 describes use of the **System Controller functions**, including functions that configure the system clock and sleep operations.
  - Chapter 4 describes use of the **Analogue Peripheral functions**, used to control the ADC, DACs and comparators.
  - Chapter 5 describes use of the **DIO functions**, used to control the 21 general-purpose digital input/output pins.
  - Chapter 6 describes use of the **UART functions**, used to control the two 16550-compatible UARTs.
  - Chapter 7 describes use of the **Timer functions**, used to control the general-purpose timers.
  - Chapter 8 describes use of the **Wake Timer functions**, used to control the wake timers that can be employed to time sleep periods.
  - Chapter 9 describes use of the **Tick Timer functions**, used to control the high-precision hardware timer.
  - Chapter 10 describes use of the **Watchdog Timer functions (JN5148 only)**, used to control the watchdog that allows software lock-ups to be avoided.

- Chapter 11 describes use of the **Pulse Counter functions (JN5148 only)**, used to control the two pulse counters.
- Chapter 12 describes use of the **Serial Interface (SI) functions**, used to control a 2-wire SI master (JN5139 and JN5148) and SI slave (JN5148 only).
- Chapter 13 describes use of the **Serial Peripheral Interface (SPI) functions**, used to control the master interface to the SPI bus.
- Chapter 14 describes use of the **Intelligent Peripheral (IP) Interface functions**, used to control the IP interface (acts as a SPI slave).
- Chapter 15 describes use of the **Digital Audio Interface (DAI) functions (JN5148 only)**, used to control the interface to an external audio device.
- Chapter 16 describes use of the **Sample FIFO Interface functions (JN5148 only)**, used to control the optional FIFO buffers between the CPU and the DAI.
- Chapter 17 describes use of the **Flash Memory functions**, used to manage the external Flash memory.

■ Part II: Reference Information comprises 16 chapters:

- Chapter 18 details the **General functions** of the API, including the API initialisation function.
- Chapter 19 details the **System Controller functions**, including functions that configure the system clock and sleep operations.
- Chapter 20 details the **Analogue Peripheral functions**, used to control the ADC, DACs and comparators.
- Chapter 21 details the **DIO functions**, used to control the 21 general-purpose digital input/output pins.
- Chapter 22 details the **UART functions**, used to control the two 16550-compatible UARTs.
- Chapter 23 details the **Timer functions**, used to control the general-purpose. timers.
- Chapter 24 details the **Wake Timer functions**, used to control the wake timers that can be employed to time sleep periods.
- Chapter 25 details the **Tick Timer functions**, used to control the high-precision hardware timer.
- Chapter 26 details the **Watchdog Timer functions (JN5148 only)**, used to control the watchdog that allows software lock-ups to be avoided.
- Chapter 27 details the **Pulse Counter functions (JN5148 only)**, used to control the two pulse counters.
- Chapter 28 details the **Serial Interface (SI) functions**, used to control a 2-wire SI master (all chips) and SI slave (JN5148 only).
- Chapter 29 details the **Serial Peripheral Interface (SPI) functions**, used to control the master interface to the SPI bus.
- Chapter 30 details the **Intelligent Peripheral (IP) Interface functions**, used to control the IP interface (acts as a SPI slave).

- Chapter 31 details the **Digital Audio Interface (DAI) functions (JN5148 only)**, used to control the interface to an external audio device.
- Chapter 32 details the **Sample FIFO Interface functions (JN5148 only)**, used to control the optional FIFO buffer between the CPU and the DAI.
- Chapter 33 details the **Flash Memory functions**, used to manage the external Flash memory.

- Part III: Appendices provides information on handling interrupts from the peripheral devices.

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.

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)
IFG	Inter-Frame Gap
IP	Intelligent Peripheral
LPRF	Low-Power Radio Frequency
MAC	Medium Access Control
NVM	Non-Volatile Memory
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-DS-JN5148	JN5148 Data Sheet
JN-DS-JN5139	JN5139 Data Sheet

Feedback Address

If you wish to comment on this manual, 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
NXP Laboratories UK Ltd
Furnival Street
Sheffield S1 4QT
United Kingdom

doc@jennic.com

Part I:
Concept and Operational Information

1. Overview

This chapter introduces the JN51xx Integrated Peripherals Application Programming Interface (API) that is used to interact with peripherals on the NXP JN5148 and JN5139 microcontrollers.

1.1 JN5148/JN5139 Integrated Peripherals

The JN5148 and JN5139 microcontrollers each feature a number of on-chip peripherals that can be used by a user application which runs on the CPU of the microcontroller. These 'integrated peripherals' are listed below. Not all of the listed peripherals are included on both JN51xx devices - where a peripheral is featured only on a certain device, this is indicated.

- System Controller
- Analogue Peripherals:
  - Analogue-to-Digital Converter (ADC)
  - Digital-to-Analogue Converters (DACs)
  - Comparators
- Digital Inputs/Outputs (DIOs)
- 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]
- Interface to external Flash memory

The above peripherals are illustrated in Figure 1 for JN5148 and Figure 2 for JN5139.

For hardware details of these peripherals, refer to the relevant chip data sheet - the *JN5148 Data Sheet (JN-DS-JN5148)* or the *JN5139 Data Sheet (JN-DS-JN5139)*.

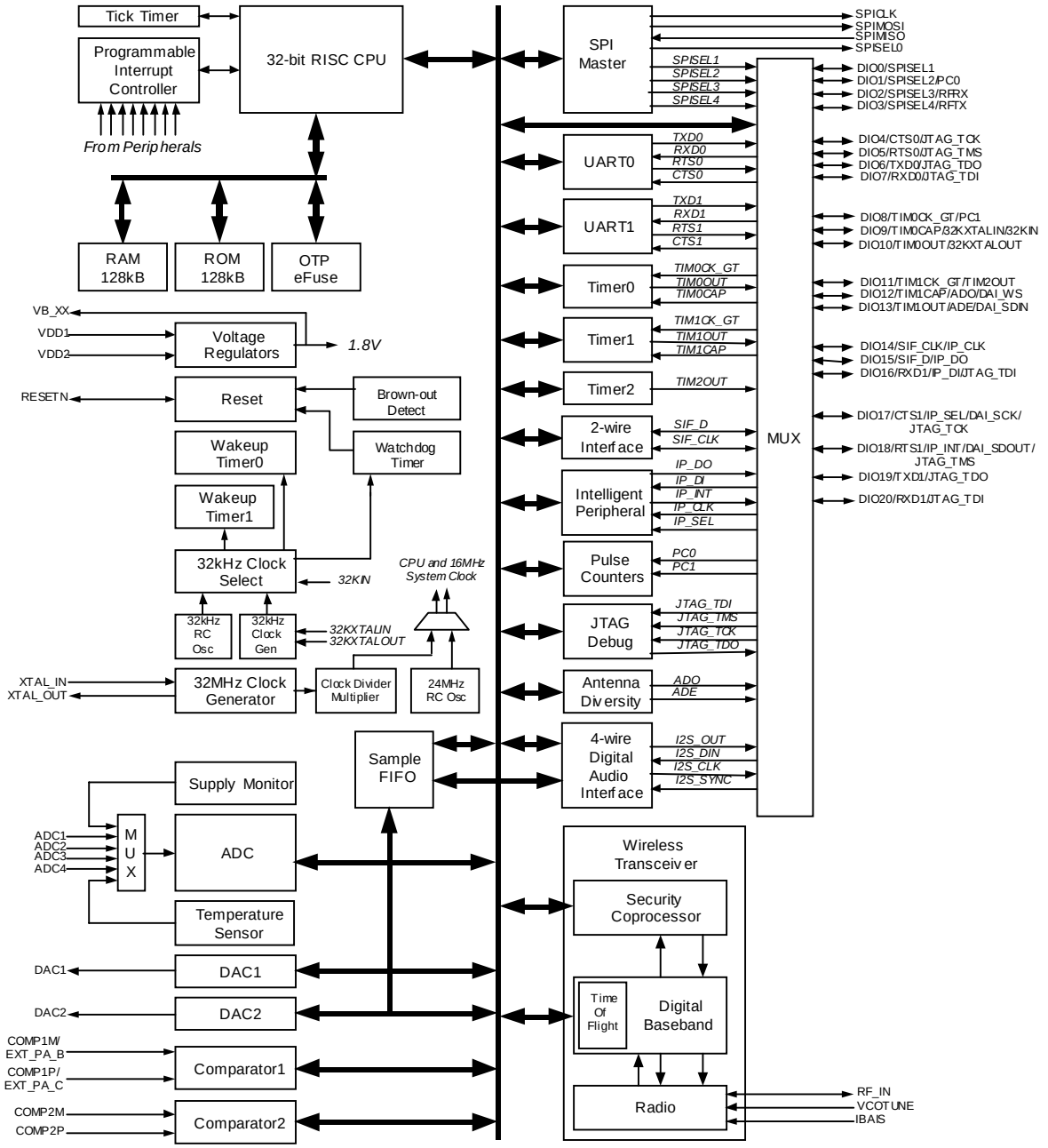


Figure 1: JN5148 Block Diagram

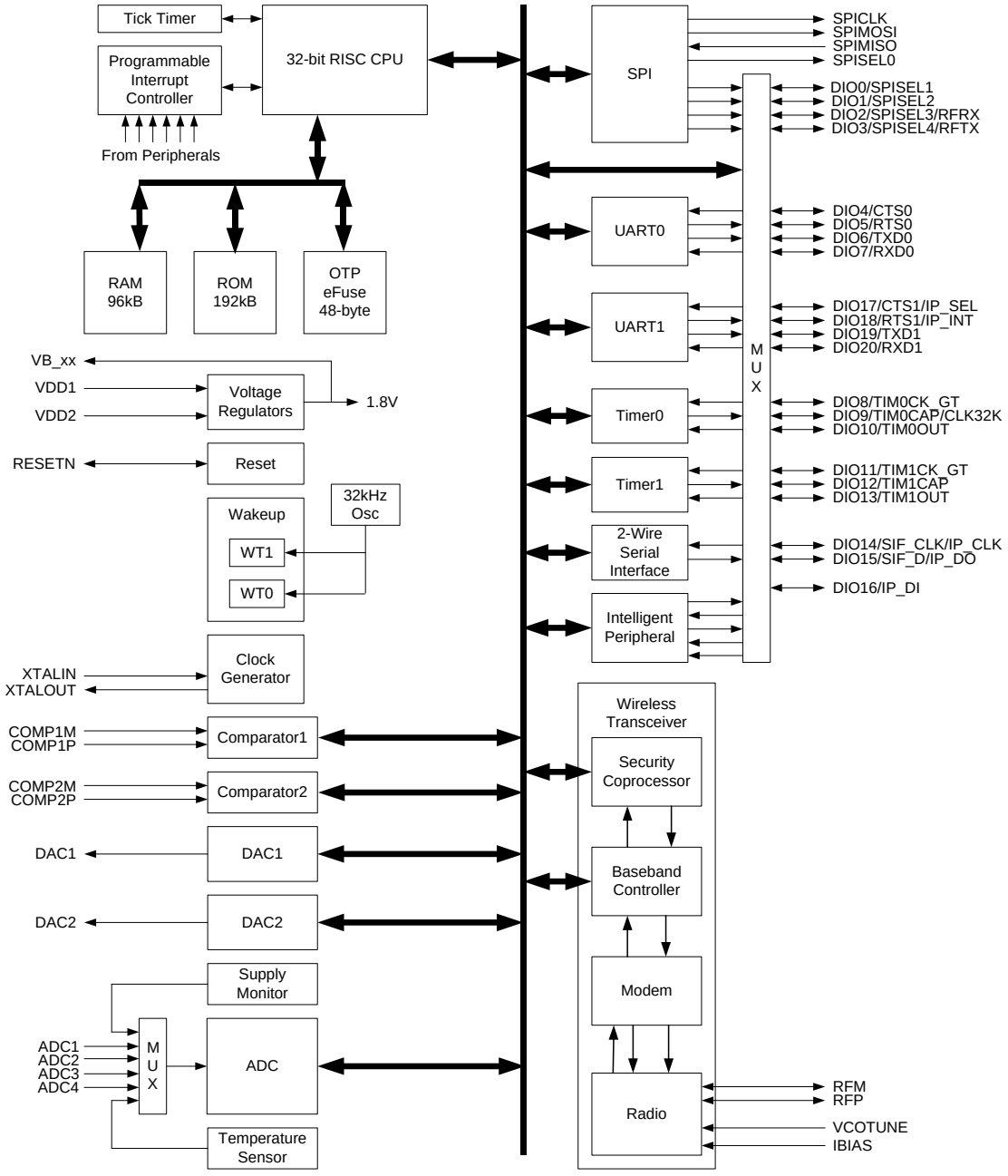


Figure 2: JN5139 Block Diagram

1.2 JN51xx Integrated Peripherals API

The JN51xx Integrated Peripherals API is a collection of C functions that can be incorporated in application code that runs on a JN5148 or JN5139 microcontroller in order to control the on-chip peripherals listed in Section 1.1. This API (sometimes referred to as the AHI) is defined in the header file **AppHardwareApi.h**, which is included in the JN51xx SDK Libraries (JN-SW-4040 for JN5148, JN-SW-4030 for JN5139). The software that is invoked by this API is located in the on-chip ROM.

This API provides a thin software layer above the on-chip registers used to control the integrated peripherals. By encapsulating several register accesses into one function call, the API simplifies use of the peripherals without the need for a detailed knowledge of their operation.

***Caution:** The Integrated Peripherals API functions are not re-entrant. A function must be allowed to complete before the 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 JN51xx device - this hardware is controlled by the wireless network protocol stack software (which may be an IEEE 802.15.4, ZigBee, JenNet or 6LoWPAN/JenNet stack), and APIs for this purpose are provided with the appropriate stack software product.
- resources of the JN51xx evaluation kit boards, such as sensors and display panels (although the buttons and LEDs on the evaluation kit boards are connected to the DIO pins of the JN51xx device) - a special function library, called the LPRF Board API, is provided by NXP for this purpose and is described in the *LPRF Board API Reference Manual (JN-RM-2003)*.

1.3 Using this Manual

The remainder of this manual is largely organised as one chapter per peripheral block. You should use the manual as follows:

1. First study Chapter 2 which describes the general functions that are not associated with one particular peripheral block. This chapter explains how to initialise the Integrated Peripherals API for use in your application code.
2. Next study Chapter 3 which describes the range of features associated with the System Controller. You may need to use one or more of these features in your application.
3. Then study those chapters in Part I: Concept and Operational Information which correspond to the particular peripherals that you wish to use in your application.

For full details of the referenced API functions, refer to Part II: Reference Information. Also note that interrupt handling is described in Part III: Appendices.

2. General Functions

This chapter describes use of the 'general functions' that are not associated with any of the peripheral blocks but may be needed in your application code (the API initialisation function will definitely be needed).

These functions cover the following areas:

- API initialisation (Section 2.1)
- Configuration of the radio transceiver (Section 2.2)
- Use of the random number generator (Section 2.3)

A function for detecting a data-stack overflow is also provided.

2.1 API Initialisation

Before calling any other function from the Integrated Peripherals API, the function **u32AHI_Init()** must be called to initialise the API. This function must be called after every reset and wake-up (from sleep) of the JN51xx microcontroller.

Caution: *If you are using JenOS (Jennic Operating System), you must not call* ***u32AHI_Init()*** *explicitly in your code, as this function is called internally by JenOS. This applies principally to users who are developing ZigBee PRO applications.*

2.2 Radio Configuration

The radio transceiver of a JN5148 or JN5139 microcontroller can be configured in a number of ways, as described in the sub-sections below.

2.2.1 Radio Transmission Power

The radio transmission power of a JN5148 or JN5139 device can be varied, the exact power range depending on the device type and, more critically, the module type (standard or high-power) on which the device sits. As a general rule:

- A standard module has a transmission power range of:
  - -32 to +2.5 dBm if JN5148-based
  - -30 to +1.5 dBm if JN5139-based
- A high-power module has a transmission power range of:
  - -16.5 to +18 dBm if JN5148-based
  - -7 to +17.5 dBm if JN5139-based

The transmission power can be set using the function **bAHI_PhyRadioSetPower()**. This function allows you to set the power to one of four (JN5148) or six (JN5139) possible levels in the power range - for details of these levels, refer to the function description in Chapter 18.

Note that:

- **bAHI_PhyRadioSetPower()** should only be called after the function **vAHI_ProtocolPower()** has been called to enable the protocol power domain - see Section 3.2.1.
- The radio transceiver of a high-power module must be explicitly enabled before it can be used - see Section 2.2.2.

Tip: The radio transmission power of a standard JN5139 module can be increased by 1.5 dBm - this is called Boost mode. Beware that this mode results in increased current consumption. Boost mode can be enabled using the function **vAppApiSetBoostMode()** which, if used, must be the first function called in your code since the setting takes effect only when the JN5139 device is initialised.

2.2.2 High-Power Modules

If a JN5148 or JN5139 high-power module is to be used, its radio transceiver must be enabled via the function **vAHI_HighPowerModuleEnable()** before attempting to operate the module. Note that the receiver and transmitter parts must both be enabled at the same time (even if you are only going to use one of them). The above function sets the CCA (Clear Channel Assessment) threshold to suit the gain of the attached high-power module.

Note: The radio transmission power of a high-power module can be set using the function **bAHI_PhyRadioSetPower()** - refer to Section 2.2.1.

***Caution:** A JN51xx high-power module cannot be used in channel 26 of the 2.4-GHz band.*

The European Telecommunications Standards Institute (ETSI) dictates a power limit for Europe of +10 dBm EIRP. You can operate a JN5148 high-power module close to this power limit by calling the function **vAHI_ETSIHighPowerModuleEnable()** after enabling the module.

2.2.3 Over-Air Transmission Properties (JN5148 Only)

The Integrated Peripherals API contains functions for the JN5148 device that allow certain over-air transmission characteristics to be deviated from the default settings dictated by the IEEE 802.15.4 protocol standard:

- **vAHI_BbcSetHigherDataRate()** can be used to increase the data-rate of over-air transmissions from the default 250 kbps to 500 or 666 kbps. These alternative rates allow on-demand burst transmissions between nodes, but performance will be degraded by at least 3 dB. The data-rate set does not only apply to data transmission but also to data reception - the device will only be able to receive data sent at the configured rate and this must be taken into account by the sending device.
- **vAHI_BbcSetInterFrameGap()** can be used to set the long Inter-Frame Gap (IFG) for the over-air radio transmission of IEEE 802.15.4 frames. The standard long IFG is 640 µs. Reducing it may result in an increase in the throughput of frames. The recommended minimum is 192 µs and the function allows a setting no lower than 184 µs. If needed, this function must be called after the radio section of the JN5148 chip has been initialised (which is done when the protocol stack is started).

If used, the above functions must be called after **vAHI_ProtocolPower()** - refer to Section 3.2.1.

Following the new data-rate and/or long IFG settings, data can be sent/received using the normal method. To later revert to standard IEEE 802.15.4 behaviour, the data-rate should be set back to 250 kbps and the long IFG should be set back to 640 µs.

2.3 Random Number Generator (JN5148 Only)

The JN5148 device features a random number generator which can produce 16-bit random numbers in one of two modes:

- **Single-shot mode:** The generator produces one random number and stops.
- **Continuous mode:** The generator runs continuously and generates a new random number every 256 µs.

The random number generator can be started in either of the above modes using the function **vAHI_StartRandomNumberGenerator()**. This function also allows an interrupt to be enabled which is produced when a random number becomes available - this is handled as a System Controller interrupt by the callback function registered using the function **vAHI_SysCtrlRegisterCallback()** (see Section 3.5).

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 of the following methods:

- Waiting for a random number generator interrupt, if enabled (see above)
- Periodically calling the function **bAHI_RndNumPoll()** to poll for the availability of a new random value

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

Note: The random number generator uses the 32-kHz clock domain (see Section 3.1) 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.

3. System Controller

This chapter describes use of the functions that control features of the System Controller.

These functions cover the following areas:

- Clock management (Section 3.1)
- Power management (Section 3.2)
- Voltage brownout (Section 3.3)
- Chip reset (Section 3.4)
- Interrupts (Section 3.5)

3.1 Clock Management

The System Controller provides clocks to the JN51xx microcontroller and is divided into two main blocks - a 16-MHz domain and a 32-kHz domain.

16-MHz Domain

The 16-MHz clock domain is used produce a system clock to run the CPU and most peripherals when the chip is fully operational. The clock for this domain is sourced as follows, dependent on the chip type:

- **JN5148:** External 32-MHz crystal oscillator or internal 24-MHz RC oscillator
- **JN5139:** External 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.

The domain normally produces a 16-MHz system clock from this clock source. However, for the JN5148 device, the system clock and CPU clock options are flexible. System clock and CPU clock configuration for the JN5148 are described in Section 3.1.1 and Section 3.1.2 respectively.

32-kHz Domain

The 32-kHz clock domain is mainly used during low-power sleep states (but also for the random number generator on the JN5148 device - see Section 2.3). While in Sleep mode (see Section 3.2.3), the CPU does not run and relies on an interrupt to wake it. The interrupt can be generated by an on-chip wake timer (see Chapter 8) or alternatively from an external source via a DIO pin (see Chapter 5), an on-chip comparator (see Section 4.3) or an on-chip pulse counter (JN5148 only - see Chapter 11). The wake timers are driven from the 32-kHz domain. The 32-kHz clock for this domain 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

Source clock selection for this domain is described in Section 3.1.4.

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 module is connected to DIO9 (pin 50).

The 32-kHz domain is still active when the chip is operating normally and can be calibrated against the 16-MHz clock to improve timing accuracy - see Section 8.2.

3.1.1 System Clock Selection (JN5148 Only)

On the JN5148 device, the function **vAHI_SelectClockSource()** is used to select the source for the system clock as either the 32-MHz crystal oscillator or the 24-MHz RC oscillator. The source clock frequency is halved to produce a system clock of 16 MHz or 12 MHz, although it is possible to configure other related frequencies (see Section 3.1.2). The above function also allows the crystal oscillator to be powered down when the RC oscillator is selected, in order to save power. Note that the identity of the current source clock can be obtained by calling the function **bAHI_GetClkSource()**.

It is important to note the following limitations while using the RC oscillator:

- The RC oscillator will produce a system clock of frequency 12 MHz to an accuracy of ±30% (unless calibrated).
- The full system cannot be run while using the RC oscillator - it is possible to execute code but it is not possible to transmit or receive. Also, calculated baud rates and timing intervals for the UARTs and timers must be based on 12 MHz.
- Switching from the crystal oscillator to the RC oscillator is not recommended.

The RC oscillator is normally only used at device wake-up (from sleep) as a temporary source clock until the crystal oscillator is properly up and running - see Section 3.1.3.

3.1.2 CPU Clock Frequency Selection (JN5148 Only)

On the JN5148 device, the default CPU clock frequency is 16 MHz or 12 MHz. However, alternative CPU clock frequencies can be configured using the function **bAHI_SetClockRate()**. A division factor (1, 2, 4 or 8) must be specified for dividing down the system source clock (32-MHz or 24-MHz) to produce the CPU clock. Thus:

- If the system clock is sourced from the 32-MHz crystal oscillator, the possible CPU clock frequencies are 4, 8, 16 and 32 MHz.
- If the system clock is sourced from the 24-MHz RC oscillator, the possible CPU clock frequencies are 3, 6, 12 and 24 MHz.

3.1.3 System Clock Start-up following Sleep (JN5148 Only)

If the 32-MHz crystal oscillator is used as the system clock source for the JN5148 device, this clock source is powered down during sleep and takes some time to become available again when the 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 crystal oscillator when it becomes available. This allows initial processing at wake-up to proceed before the crystal oscillator is ready.

The function **vAHI_EnableFastStartUp()** can be called before going to sleep to ensure that the RC oscillator will be used immediately on waking. The subsequent switch to the crystal oscillator can be either automatic or manual:

- **Automatic switch:** The crystal oscillator starts immediately on waking from sleep, allowing it to warm up and stabilise while the boot code is running. This oscillator is then automatically and seamlessly switched to when ready. The function **bAHI_GetClkSource()** can be used to determine whether the switch has taken place.
- **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, this 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 RC oscillator is being used, you should not attempt to transmit or receive, and you can only use the JN5148 peripherals with special care (see Section 3.1.1). 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.

3.1.4 32-kHz Clock Selection

As stated in the introduction to Section 3.1, a choice of source for the 32-kHz clock is available on the JN5139 and JN5148 devices. The selection of this source clock is detailed separately below for the two devices.

Note: On both the JN5139 and JN5148 devices, the default clock source is the internal 32-kHz RC oscillator. The functions described below only need to be called if an external 32-kHz clock source is required. Once an external source has been selected, it is not possible to switch back to the internal RC oscillator.

JN5139 Clock Selection

On the JN5139 device, the 32-kHz clock can be optionally sourced from an external clock module. If this external clock source is required, the function **vAHI_ExternalClockEnable()** must be called. This function should be called only following device start-up/reset and not following wake-up from sleep. Once this function has been called to enable an external clock input, you are not advised to subsequently change back to the internal RC oscillator.

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

JN5148 Clock Selection

On the JN5148 device, the 32-kHz clock can be optionally sourced from an external clock module (RC circuit) or an external crystal oscillator. If one of these external clock sources is required, the function **bAHI_Set32KhzClockMode()** must be called. If required, this function should be called near the start of the application.

If selecting the external crystal oscillator then **bAHI_Set32KhzClockMode()** 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. Note that the external crystal can take up to one second to start.

The connections to the external clock source must be made as follows:

- The external clock module must be supplied on DIO9 (pin 50). You must first disable the pull-up on DIO9 using the function **vAHI_DioSetPullup()**.
- The external crystal oscillator must be attached on DIO9 (pin 50) and DIO10 (pin 51). The pull-ups on DIO9 and DIO10 are disabled automatically.

Note that there is no need to explicitly configure DIO9 or DIO10 as an input, as this is done automatically by **bAHI_Set32KhzClockMode()**.

3.2 Power Management

This section describes how to control the power to a JN51xx microcontroller using the Integrated Peripherals API. This includes control of the power regulator that supplies certain on-chip peripherals and the management of low-power sleep modes.

3.2.1 Power Domains

A JN51xx microcontroller has a number of independent power domains, supplied by separate voltage regulators, as follows:

- **Digital Logic domain:** This domain supplies the CPU and digital peripherals as well as the modem, encryption coprocessor and baseband controller. The clock for this domain can be enabled/disabled by the application (see Section 3.2.2). The domain is always unpowered during sleep.
- **Analogue domain:** This domain supplies the ADC and DACs. The regulator is switched on when the function **vAHI_ApConfigure()** is called to configure the analogue peripherals - see Chapter 4. The domain is always unpowered during sleep.
- **RAM domain:** This domain supplies the on-chip RAM. The domain may be powered or unpowered during sleep.
- **Radio domain:** This domain supplies the radio transceiver. The domain is always unpowered during sleep.
- **VDD Supply domain:** This domain supplies the wake timers, DIO blocks, comparators and 32-kHz oscillators. The domain is driven from the external supply (battery) and is always powered. However, the wake timers and 32-kHz oscillators may be powered or unpowered during sleep.

The separate voltage regulators for the CPU (Digital Logic domain) and on-chip RAM provide flexibility in implementing different low-power sleep modes, allowing the memory to be either powered (and its contents maintained) or unpowered while the CPU is powered down - for further information on sleep modes, refer to Section 3.2.3.

3.2.2 Digital Logic Domain Clock

The clock for the Digital Logic domain can be enabled/disabled using the function **vAHI_ProtocolPower()**, but disabling this clock outside of a reset or sleep cycle must be done with caution. The following points should be noted:

- Disabling the Digital Logic domain clock leaves the clock powered but disabled (gated).
- Disabling the Digital Logic domain clock causes the IEEE 802.15.4 MAC settings to be lost. Therefore, you must save the current MAC settings before disabling the clock. On re-enabling clock, the MAC settings must be restored from the saved settings. You can save and restore the MAC settings using functions of the 802.15.4 Stack API, described in the *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*:
  - To save the MAC settings, use the function **vAppApiSaveMacSettings()**.
  - To restore the saved MAC settings, use the function **vAppApiRestoreMacSettings()** - the Digital Logic domain clock is automatically re-enabled, since this function calls **vAHI_ProtocolPower()**.
- Do not call **vAHI_ProtocolPower()** to disable the Digital Logic domain clock while the 802.15.4 MAC layer is active, otherwise the microcontroller may freeze.
- While the Digital Logic domain clock is disabled, do 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.

3.2.3 Low-Power Modes

The JN51xx microcontrollers are able to enter a number of low-power modes in order to conserve power during periods when the device does not need to be fully active. Generally, there are two low-power modes, Sleep mode (including Deep Sleep) and Doze mode, described below.

Sleep and Deep Sleep Modes

In Sleep mode, most of the internal chip functions are shut down to save power, including the CPU and the majority of on-chip peripherals. However, the states of the DIO pins are retained, including the output values and pull-up enables, which preserves any interface to the outside world. In addition, the DAC outputs are put into a high-impedance state. The on-chip RAM, the 32-kHz oscillator, the comparators and the pulse counters (JN5148 only) can optionally remain active during sleep.

Sleep mode is started using the function **vAHI_Sleep()**, when one of four sleep modes can be selected which depend on whether RAM and the 32-kHz oscillator are to be powered off. The significance of the 32-kHz oscillator and RAM during sleep is outlined below:

- **32-kHz Oscillator:** The 32-kHz oscillator (internal RC, external clock or external crystal) can be either left running or stopped for the duration of sleep. This oscillator is used by the wake timers and must be left running if a wake timer will be used to wake the device from sleep. Otherwise, the oscillator should be switched off during sleep. However, if an external source is used for this oscillator, it is not recommended that the oscillator is stopped on entering sleep mode.

Note: On the JN5148 device, if a pulse counter is to be run with debounce while the device is asleep, the 32-kHz oscillator must be left running - see Chapter 11.

- **On-chip RAM:** Power to on-chip RAM can be either maintained or removed during sleep. The application program, stack context data and application data are all held in on-chip RAM while the microcontroller is fully active, but are lost if the power to RAM is switched off.
  - If the power to RAM is removed during sleep, the application is re-loaded into RAM from external Non-Volatile Memory (NVM) on exiting sleep mode - stack context and application data may also be re-loaded by the application, if they were saved to NVM before entering sleep mode.
  - If the power to RAM is maintained during sleep, the application and data will be preserved. This option is useful for short sleep periods, when the time taken on waking to re-load the application and data from NVM to RAM is significant compared with the sleep duration.

A further low-power option is Deep Sleep mode in which the CPU, RAM and both the 16-MHz and 32-kHz clock domains are powered down. In addition, external NVM is also powered down during Deep Sleep mode. This option obviously provides a bigger power saving than Sleep mode.

Note: External NVM is not powered down during normal Sleep mode. If required, you can power down a Flash memory device using **vAHI_FlashPowerDown()**, which must be called before **vAHI_Sleep()**, provided you are using a compatible Flash device. For full details, refer to Section 17.4.

The microcontroller can subsequently be woken from Sleep mode by one of the following:

- DIO interrupt (see Chapter 5)
- Wake timer interrupt (needs 32-kHz oscillator to be running - see Chapter 8)
- Comparator interrupt (see Section 4.3)
- Pulse counter interrupt (JN5148 only - see Chapter 11)

The device can only be woken from Deep Sleep mode by its reset line being pulled low (all chips) or by an external event which triggers a change on a DIO pin.

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.

Doze Mode

Doze mode is a low-power mode in which the CPU, RAM, radio transceiver and digital peripherals remain powered but the clock to the CPU is stopped (all other clocks continue as normal). This mode provides less of a power saving than Sleep mode but allows a quicker recovery back to full working mode. Doze mode is useful for very short periods of low power consumption - for example, while waiting for a timer event or for a transmission to complete.

The CPU can be put into Doze mode by calling the function **vAHI_CpuDoze()**. It is subsequently brought out of Doze mode by almost any interrupt - note that a Tick Timer interrupt can be used to bring the CPU out of Doze mode on the JN5148 device but not on the JN5139 device.

3.2.4 Power Status

The power status of the JN51xx microcontroller can be obtained using the function **u8AHI_PowerStatus()**. This function returns a bitmap in which the individual bits indicate whether:

- The device has completed a sleep-wake cycle
- RAM contents were retained during sleep
- The analogue power domain is switched on
- The protocol logic is operational - clock is enabled

3.3 Voltage Brownout (JN5148 Only)

A ‘brownout’ is a fall in the supply voltage to a device or system below a pre-defined level, which may hinder or be harmful to the operation of the device/system. The JN5148 microcontroller is equipped with a brownout detection feature which can be configured and monitored through functions of the Integrated Peripherals API.

3.3.1 Configuring Brownout Detection

By default on the JN5148 device, brownout detection is automatically enabled and the brownout voltage is set to 2.3V. On detection of a brownout, the chip will be automatically be reset.

The above brownout settings can be changed by calling the function **vAHI_BrownOutConfigure()**, which allows the configuration of the following:

- **Brownout detection:** The brownout detection feature can be enabled/disabled - if the configuration function is called and brownout detection is required, the feature must be explicitly enabled in the function.
- **Brownout level:** The brownout voltage level can be set to one of four values - 2.0V, 2.3V, 2.7V or 3.0V.
- **Reset on brownout:** The automatic reset on the occurrence of a brownout can be enabled/disabled.
- **Brownout interrupts:** Two separate interrupts relating to brownout can be enabled/disabled:
  - An interrupt can be generated when the device enters the brownout state (supply voltage falls below the brownout voltage level).
  - An interrupt can be generated when the device leaves the brownout state (supply voltage rises above the brownout voltage level).

After the return of the configuration function, there will be a delay of up the 30 µs before the new settings take effect.

Note: Following a device reset or sleep, the default brownout settings are re-instated.

3.3.2 Monitoring Brownout

Provided that brownout detection is enabled (see Section 3.3.1), the brownout status of the JN5148 device can be monitored in one of three ways: automatic reset, interrupts or polling. These options are described below.

Automatic Reset on Brownout

An automatic reset on a brownout is enabled by default, but can also be enabled/ disabled through the function **vAHI_BrownOutConfigure()**. Following a chip reset, the application can check whether a brownout was the cause of the reset by calling the function **bAHI_BrownOutEventResetStatus()**.

Brownout Interrupts

Interrupts can be generated when the device enters the brownout state and/or when it exits the brownout state. These two interrupts can be individually enabled/disabled through the function **vAHI_BrownOutConfigure()**. Brownout interrupts are System Controller interrupts and are handled by the callback function registered using the function **vAHI_SysCtrlRegisterCallback()** - see Section 3.5.

Polling for Brownout

If brownout interrupts and automatic reset are disabled (but detection is still enabled), the brownout state of the device can be obtained by manually polling via the function **u32AHI_BrownOutPoll()**. This function will indicate whether the supply voltage is currently above or below the brownout level.

3.4 Resets

The JN51xx microcontroller can be reset from the application using the function **vAHI_SwReset()**. This function initiates the full reset sequence for the chip and is the equivalent of pulling the external RESETN line low. Note that during a chip reset, the contents of on-chip RAM are likely to be lost.

One or more external devices may also be connected to the RESETN line. This line can be pulled low without resetting the chip by calling the function **vAHI_DriveResetOut()**. This function allows you to specify the length of time for which the line will be held low. Thus, any external devices connected to this line may be affected.

Note: An external RC circuit can be connected to the RESETN line in order to generate a reset. The required resistance and capacitance values are specified in the data sheet for your microcontroller.

3.5 System Controller Interrupts

System Controller interrupts cover a number of on-chip peripherals that do not have their own interrupts:

- Comparators
- DIOs
- Wake Timers
- Pulse Counter (JN5148 only)
- Random Number Generator (JN5148 only)
- Brownout detector (JN5148 only)

Interrupts for these peripherals can be individually enabled using their own functions from the Integrated Peripherals API.

The handling of interrupts from these sources must be incorporated in a user-defined callback function, registered using the function **vAHI_SysCtrlRegisterCallback()**. The registered callback function is automatically invoked when an interrupt of the type E_AHI_DEVICE_SYSCTRL occurs. The exact source of the interrupt (from the peripherals listed above) can then be identified from a bitmap that is passed into the function. Note that the interrupt will be automatically cleared before the callback function is invoked.

Note: The callback function prototype is detailed in Appendix A.1. The interrupt source information is provided in Appendix B.

Caution: *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.*

4. Analogue Peripherals

This chapter describes control of the analogue peripherals using functions of the Integrated Peripherals API.

The are three categories of analogue peripheral on the JN51xx microcontrollers:

- Analogue-to-Digital Converter [ADC] (Section 4.1)
- Digital-to-Analogue Converter [DAC] (Section 4.2)
- Comparator (Section 4.3)

Analogue peripheral interrupts are described in Section 4.4.

Note: The ADC and DACs are located in the same peripheral block. They can be used independently of each other or in any combination. When used concurrently, they operate to common timings.

4.1 ADC

The JN51xx microcontrollers each include a 12-bit Analogue-to-Digital Converter (ADC). This device samples an analogue input signal to produce a 12-bit digital representation of the input voltage. The ADC samples the input voltage at one instant in time and holds this voltage (in a capacitor) while converting it to a 12-bit binary value - the total sample/convert duration is called the conversion time.

The ADC may sample periodically to produce a sequence of 12-bit values representing the behaviour of the input voltage over time. The rate at which the sampling events take place is called the sampling frequency. According to the Nyquist sampling theorem, the sampling frequency must be at least twice the highest frequency to be measured in the input signal. If the input signal contains frequencies of more than half the sampling frequency, these frequencies will be aliased. To prevent aliasing, a low-pass filter should be applied to the ADC input in order to remove frequencies greater than half the sampling frequency.

The ADC can take its analogue input from an external source, an on-chip temperature sensor and an internal voltage monitor (see below). The input voltage range is also selectable as between zero and a reference voltage, or between zero and twice this reference voltage (see below).

Note: The function **vAHI_ApConfigure()**, referred to below, is used to configure properties that apply to the ADC and DACs.

When using the ADC, the first analogue peripheral function to be called must be **vAHI_ApConfigure()**, which allows the following properties to be configured:

- **Clock:**

  The clock input for the ADC is provided by the 16-MHz on-chip clock, which is divided down. The target frequency is selected using **vAHI_ApConfigure()** (this clock output is shared with the DACs). The recommended target frequency for the ADC is 500 kHz.

- **Sampling interval and conversion time:**

  The sampling interval determines the time over which the ADC will integrate the analogue input voltage before performing the conversion - in fact, the integration occurs over three times this interval (see Figure 3). This interval is set as a multiple of the ADC clock period (2x, 4x, 6x or 8x), where this multiple is selected using **vAHI_ApConfigure()**. Normally, it should be set to 2x - for details, refer to the data sheet for your microcontroller.

  The time allowed to perform the subsequent conversion is 14 clock periods. Thus, the total time to sample and convert (the conversion time) is given by:

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

  For a visual illustration, refer to Figure 3.

- **Reference voltage:**

  The permissible range for the analogue input voltage is defined relative to a reference voltage V_{ref}, which can be sourced internally or externally. The source of V_{ref} is selected using **vAHI_ApConfigure()**.

  The input voltage range can be selected as either 0-V_{ref} or 0-2V_{ref}, which is selected the **vAHI_AdcEnable()** function - see later.

- **Voltage regulator:**

  In order to minimise the amount of digital noise in the ADC, the device is powered (along with the DACs) from a separate voltage regulator, sourced from the analogue supply VDD1. The regulator (and therefore power) can be enabled/disabled using **vAHI_ApConfigure()**. Once enabled, it is necessary to wait for the regulator to stabilise - the function **bAHI_APRegulatorEnabled()** can be used to check whether the regulator is ready.

- **Interrupt:**

  Interrupts can be enabled such that an interrupt (of the type E_AHI_DEVICE_ANALOGUE) is generated after each individual conversion. This is particularly useful for ADC continuous (periodic) conversion. Interrupts can be enabled/disabled using **vAHI_ApConfigure()**. Analogue peripheral interrupt handling is described in Section 4.4.

The ADC must then be further configured and enabled (but not started) using the function **vAHI_AdcEnable()**. This function allows the following properties to be configured.

- **Input source:**

  The ADC can take its input from one of six multiplexed sources, comprising four external pins (DIOs), an on-chip temperature sensor and an internal voltage monitor. The input is selected using **vAHI_AdcEnable()**.

- **Input voltage range:**

  The permissible range for the analogue input voltage is defined relative to the reference voltage V_{ref}. The input voltage range can be selected as either 0-V_{ref} or 0-2V_{ref} (an input voltage outside this range results in a saturated digital output). The analogue voltage range is selected using **vAHI_AdcEnable()**.

- **Conversion mode:**

  The ADC can be configured to perform conversions in the following modes:

  - **Single-shot:** A single conversion is performed (see Section 4.1.1).
  - **Continuous:** Conversions are performed repeatedly (see Section 4.1.2).
  - **Accumulation (JN5148 only):** A fixed number of conversions are performed and the results are added together (see Section 4.1.3).

  Single-shot mode or continuous mode can be selected using **vAHI_AdcEnable()**. In all three cases, the conversion time for an individual conversion is given by [(3 x *sampling interval)* + 14] x *clock period*, as illustrated in Figure 3. In the cases of continuous mode and accumulation mode, after this time the next conversion will start and the sampling frequency will be the reciprocal of the conversion time.

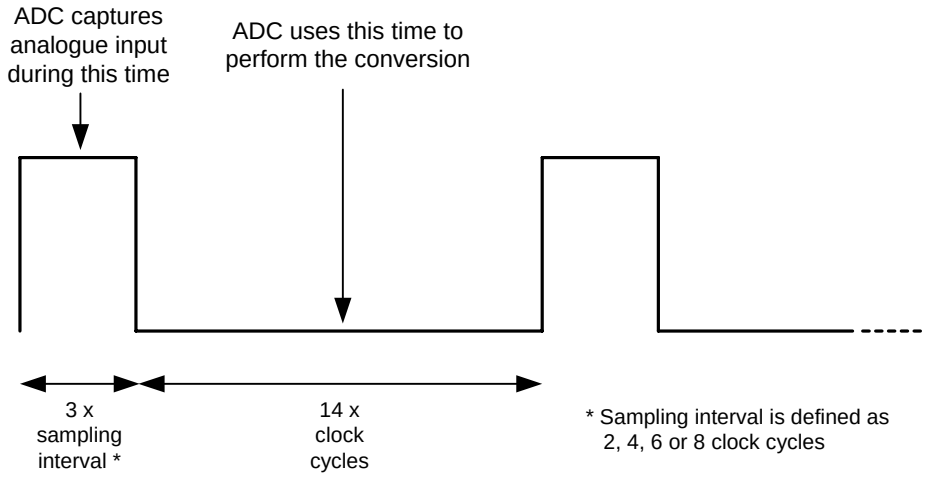


Figure 3: ADC Sampling

Once the ADC has been configured using first **vAHI_ApConfigure()** and then **vAHI_AdcEnable()**, conversion can be started in one of the available modes. Operation of the ADC in these modes is described in the subsections below:

- Single-shot mode: Section 4.1.1
- Continuous mode: Section 4.1.2
- Accumulation mode (JN5148 only): Section 4.1.3

Note that only the ADC can generate analogue peripheral interrupts (of the type E_AHI_DEVICE_ANALOGUE) - these interrupts are handled by a user-defined callback function registered via **vAHI_APRegisterCallback()**. Refer to Section 4.4 for more information on analogue peripheral interrupt handling.

4.1.1 Single-Shot Mode

In single-shot mode, the ADC performs one conversion and then stops. To operate in this way, single-shot mode must have been selected when the ADC was enabled using **vAHI_AdcEnable()**. The conversion can then be started using the function **vAHI_AdcStartSample()**.

Completion of the conversion can be detected in one of two ways:

- An interrupt can be generated on completion - in this case, analogue peripheral interrupts must have been enabled in the function **vAHI_ApConfigure()**.
- The function **bAHI_AdcPoll()** can be used to check whether the conversion has completed.

Once the conversion has been performed, the 12-bit result can be obtained using the function **u16AHI_AdcRead()**.

Caution: *The ADC cannot be used in single-shot mode while either of the DACs is enabled - see* Section 4.2*. However, it can be used in continuous or accumulation mode - see* Section 4.1.2 *below.*

4.1.2 Continuous Mode

In continuous mode, the ADC performs repeated conversions indefinitely (until stopped). To operate in this way, continuous mode must have been selected when the ADC was enabled using **vAHI_AdcEnable()**. The conversions can then be started using the function **vAHI_AdcStartSample()**.

The sampling frequency in continuous mode is given by the reciprocal of the conversion time, where:

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

Completion of an individual conversion can be detected in one of two ways:

- An interrupt can be generated on completion - in this case, analogue peripheral interrupts must have been enabled in the function **vAHI_ApConfigure()**.
- The function **bAHI_AdcPoll()** can be used to check whether the conversion has completed.

Once an individual conversion has been performed, the 12-bit result can be obtained using the function **u16AHI_AdcRead()**. The result remains available to be read by this function until the next conversion has completed.

The conversions can be stopped using the function **vAHI_AdcDisable()**.

4.1.3 Accumulation Mode (JN5148 Only)

In accumulation mode on the JN5148 device, the ADC performs a fixed number of conversions and then stops. The results of these conversions are added together to allow them to be averaged. To operate in this mode, the conversions must be started using the function **vAHI_AdcStartAccumulateSamples()**. The number of conversions is selected in this function as 2, 4, 8 or 16.

Note: When the ADC is started in accumulation mode, the conversion mode selected in **vAHI_AdcEnable()** is ignored.

The sampling frequency in accumulation mode is given by the reciprocal of the conversion time, where:

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

Completion of ALL the conversions can be detected in one of two ways:

- An interrupt can be generated on completion - in this case, analogue peripheral interrupts must have been enabled in the function **vAHI_ApConfigure()**.
- The function **bAHI_AdcPoll()** can be used to check whether the conversions have completed.

Once the conversions have been performed, the cumulative result can be obtained using the function **u16AHI_AdcRead()**. Note that this function delivers the sum of the results for individual conversions - the averaging calculation must be performed by the application (by dividing by the number of conversions).

The conversions can be stopped at any time using the function **vAHI_AdcDisable()**.

4.2 DACs

The JN51xx microcontrollers include two Digital-to-Analogue Converters (DACs), denoted DAC1 and DAC2.

- On the JN5139 device, they are 11-bit DACs
- On the JN5148 device, they are 12-bit DACs

Each DAC can take a digital value of up to 11/12 bits and, from it, produce an analogue output as a proportional voltage on a dedicated pin, also denoted DAC1 or DAC2.

The DACs share their peripheral block with the ADC and their operation is linked to that of the ADC. In particular, the ADC and DACs use the same clock, and the ADC conversion time dictates the DAC conversion time (see Section 4.1). When a DAC and the ADC are used concurrently, a DAC conversion occurs at exactly the same time as an ADC conversion.

Note 1: On the JN5139 device, only one DAC can be used at any one time, since the two DACs share resources. If both DACs are to be used concurrently, they can be multiplexed.

Note 2: When a DAC is enabled, the ADC cannot be used in single-shot mode but can be used in continuous mode.

Note 3: The function **vAHI_ApConfigure()**, referred to below, is used to configure properties that apply to the DACs and the ADC.

When using a DAC, the first analogue peripheral function to be called must be **vAHI_ApConfigure()**, which allows the following properties to be configured:

- **Clock:**

  The clock input for the DAC is provided by the 16-MHz on-chip clock, which is divided down. The target frequency is selected using **vAHI_ApConfigure()** (this clock is shared with the other DAC and the ADC).

- **Conversion time:**

  The operation of a DAC is linked to the ADC and the DAC conversion time is equal to the ADC conversion time for an individual sample, described in Section 4.1 and given by:

  $$[(3 \times \textit{sampling interval}) + 14] \times \textit{clock period}$$

  The sampling interval is selected in **vAHI_ApConfigure()** as a multiple of the DAC clock period (2x, 4x, 6x or 8x) - this setting is shared with the other DAC and ADC. The resulting analogue voltage is maintained on the output pin until the next digital value is submitted to the DAC for conversion.

- **Reference voltage:**

  The maximum range for the analogue output voltage can be defined relative to a reference voltage V_{ref}, which can be sourced internally or externally. The source of V_{ref} is selected using the function **vAHI_ApConfigure()**.

  The output voltage range can be selected as either 0-V_{ref} or 0-2V_{ref}, which is selected using the function **vAHI_DacEnable()** - see later.

- **Voltage regulator:**

  In order to minimise the amount of digital noise in the DACs, they are powered (along with the ADC) from a separate voltage regulator, sourced from the analogue supply VDD1. The separate regulator (and therefore power) can be enabled/disabled using **vAHI_ApConfigure()**. Once enabled, it is necessary to wait for the regulator to stabilise - the function **bAHI_APRegulatorEnabled()** can be used to check whether the regulator is ready.

The DAC must then be further configured and enabled using the function **vAHI_DacEnable()**. This function allows the following properties to be configured.

- **Output voltage range:**

  The maximum range for the analogue output voltage can be defined relative to a reference voltage V_{ref}. The output voltage range can be selected as either 0-V_{ref} or 0-2V_{ref}, selected using **vAHI_DacEnable()**.

- **First conversion value (JN5148 only):**

  For the JN5148 device, the first value to be converted must be specified through **vAHI_DacEnable()**. In this case, this function also starts the conversion - see below.

Starting a DAC

Starting a DAC differs between the chip types:

- On the JN5148 device, the first value to be converted is specified through the **vAHI_DacEnable()** function and the conversion starts immediately after this function call. All subsequent values to be converted must then be specified through calls to the function **vAHI_DacOutput()**.
- On the JN5139 device, all values to be converted must be specified through calls to the function **vAHI_DacOutput()**. Thus, conversion will begin after the first call to this function.

Note: The value to be converted must be specified as a 16-bit value, but only the 11/12 least significant bits are used (all other bits are ignored).

The function **bAHI_DacPoll()** can be used to check whether a DAC conversion has completed, before submitting the next value to be converted.

A DAC can be disabled using the function **vAHI_DacDisable()**.

4.3 Comparators

The JN51xx microcontrollers include two comparators, denoted COMP1 and COMP2.

A comparator can be used to compare two analogue inputs. The comparator changes its two-state digital output (high to low or low to high) when the arithmetic difference between the inputs changes sense (positive to negative or negative to positive). A comparator can be used as a basis for measuring the frequency of a time-varying analogue input when compared with a constant reference input.

Thus, each comparator has two analogue inputs. One analogue input (on the 'positive' pin COMP1P or COMP2P) carries the externally sourced signal to be monitored - the input voltage must always remain within the range 0V to V_{dd} (the chip supply voltage). This external signal will be compared with a reference signal, which can be sourced internally or externally, as follows:

- externally from the 'negative' pin COMP1M or COMP2M
- internally from the analogue output of the corresponding DAC (DAC1 or DAC2)
- internally from the reference voltage V_{ref} (the source of V_{ref} is selected using the function **vAHI_ApConfigure()**)

The reference signal is selected from the above options via the function **vAHI_ComparatorEnable()**, which is used to configure and enable the comparator.

Note 1: By default, the comparators are enabled in low-power mode. Refer to Section 4.3.2 for more details.

Note 2: Calling **vAHI_ComparatorEnable()** while the ADC is operating may lead to corruption of the ADC results. Therefore, if required, this function should be called before starting the ADC.

The comparator has two possible states - high or low. The comparator state is determined by the relative values of the two analogue input voltages - that is, by the instantaneous voltages of the signal under analysis, V_{sig}, and the reference signal, V_{refsig}. The relationships are as follows:

$$V_{sig} > V_{refsig} \Rightarrow \text{high}$$

$$V_{sig} < V_{refsig} \Rightarrow \text{low}$$

or in terms of differences:

$$V_{sig} - V_{refsig} > 0 \Rightarrow \text{high}$$

$$V_{sig} - V_{refsig} < 0 \Rightarrow \text{low}$$

Thus, as the signal levels vary with time, when V_{sig} rises above V_{refsig} or falls below V_{refsig}, the state of the comparator result changes. In this way, V_{refsig} is used as the threshold against which V_{sig} is assessed.

In reality, this method of functioning is sensitive to noise in the analogue input signals causing spurious changes in the comparator state. This situation can be improved by using two different thresholds:

- An upper threshold, V_{upper}, for low-to-high transitions
- A lower threshold, V_{lower}, for high-to-low transitions

The thresholds V_{upper} and V_{lower} are defined such that they are above and below the reference signal voltage V_{refsig} by the same amount, where this amount is called the hysteresis voltage, V_{hyst}.

That is:

$$V_{upper} = V_{refsig} + V_{hyst}$$

$$V_{lower} = V_{refsig} - V_{hyst}$$

The hysteresis voltage is selected using the **vAHI_ComparatorEnable()** function. It can be set to 0, 5, 10 or 20 mV. The selected hysteresis level should be larger than the noise level in the input signal.

The comparator two-threshold mechanism is illustrated in Figure 4 below for the case when the reference signal voltage V_{refsig} is constant.

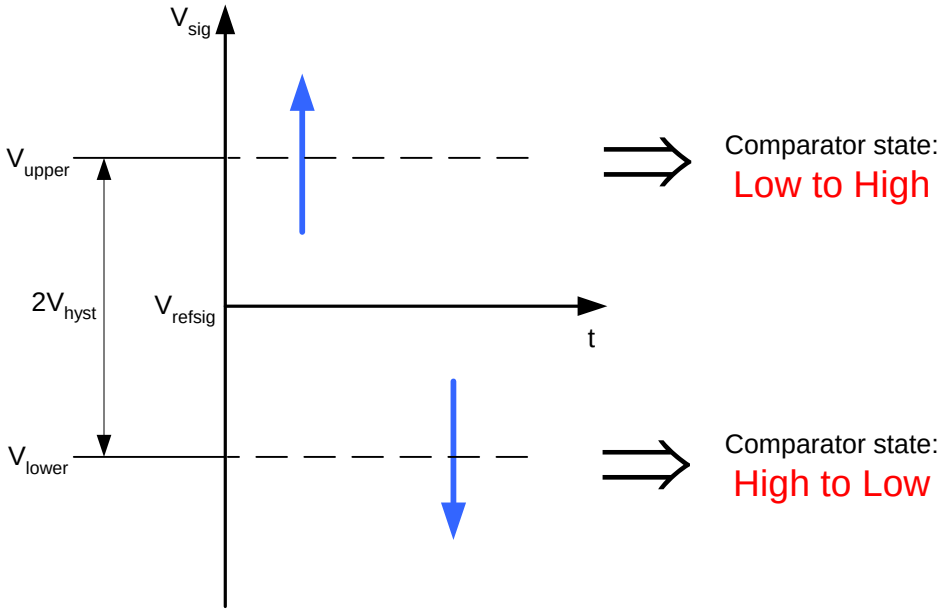


Figure 4: Upper and Lower Thresholds of Comparator

Note that there is a time delay between a change in the comparator inputs and the resulting state reported by the comparator.

As well as configuring a specified comparator, **vAHI_ComparatorEnable()** also starts operation of the comparator. The current state of the comparator (high or low) can be obtained at any time using the function **u8AHI_ComparatorStatus()**. The comparator can be stopped at any time using the function **vAHI_ComparatorDisable()**.

4.3.1 Comparator Interrupts and Wake-up

The comparators allow an interrupt to be generated on either a low-to-high or high-to-low transition. Interrupts can only be produced on transitions in one direction (and not both). Interrupts can be enabled using the function **vAHI_ComparatorIntEnable()**. The function is used to both enable/disable comparator interrupts and select the direction of the transitions that will trigger the interrupts.

Important: Comparator interrupts are System Controller interrupts and not analogue peripheral interrupts. They must therefore be handled by a callback function that is registered via **vAHI_SysCtrlRegisterCallback()**.

A comparator interrupt can be used as a signal to wake a node from sleep - this is then referred to as a 'wake-up interrupt'. To use this feature, interrupts must be configured and enabled using **vAHI_ComparatorIntEnable()**, as described above. Note that during sleep, the reference signal V_{refsig} is taken from an external source via the 'negative' pin COMP1M or COMP2M. The wake-up interrupt status can be checked using the function **u8AHI_ComparatorWakeStatus()**.

4.3.2 Comparator Low-Power Mode

The comparators are able to operate in a low-power mode, in which each comparator draws only 1.2 µA of current, compared with 70 µA when operating in standard-power mode. Comparator low-power mode can be enabled/disabled using the function **vAHI_ComparatorLowPowerMode()**, which affects both comparators together.

Low-power mode is enabled by default when a comparator is configured and started using **vAHI_ComparatorEnable()**. Therefore, to operate the comparators in standard-power mode, the function **vAHI_ComparatorLowPowerMode()** must be called to disable low-power mode.

Low-power mode is beneficial in helping to minimise the current drawn by a device that employs energy harvesting. It is also beneficial during sleep to optimise the energy conserved. The disadvantage of low-power mode is a slower response time for the comparator - that is, a longer delay between a change in the comparator inputs and the resulting state reported by the comparator. However, if the response time is not important, low-power mode should normally be used.

4.4 Analogue Peripheral Interrupts

Analogue peripheral interrupts (of the type E_AHI_DEVICE_ANALOGUE) are only generated by the ADC (the DACs do not generate interrupts and the comparators generate System Controller interrupts). The analogue peripheral interrupts are enabled in the function **vAHI_ApConfigure()** and are handled using a user-defined callback function registered using the function **vAHI_APRegisterCallback()**. For details of the callback function prototype, refer to Appendix A.1. The interrupt is automatically cleared when the callback function is invoked.

Caution: *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.*

The exact interrupt source depends on the ADC operating mode (single-shot, continuous, accumulation):

- In single-shot and continuous modes, a 'capture' interrupt will be generated after each individual conversion.
- In accumulation mode on the JN5148 device, an 'accumulation' interrupt will be generated when the final accumulated result is available.

Once an ADC result becomes available, it can be obtained by calling the function **u16AHI_AdcRead()** within the callback function.

5. Digital Inputs/Outputs (DIOs)

This chapter describes control of the Digital Inputs/Outputs (DIOs) using functions of the Integrated Peripherals API.

The JN51xx microcontrollers each include 21 general-purpose digital input/output (DIO) pins, denoted DIO0 to DIO20. Each pin can be individually configured as an input or output. However, the DIO pins are shared with the following on-chip peripherals/features:

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

A shared DIO is not available when the corresponding peripheral/feature is enabled. For details of the shared pins, refer to the data sheet for your microcontroller.

From reset, all peripherals are disabled and the DIOs are configured as inputs.

In addition to normal operation, when configured as inputs, the DIOs can be used to generate interrupts and wake the device from sleep - see Section 5.2. Note that the interrupts triggered by the DIOs are System Controller interrupts and are handled by a callback function registered via **vAHI_SysCtrlRegisterCallback()** - see Section 3.5.

5.1 Using the DIOs

This section describes how to use the Integrated Peripherals API functions to use the DIOs.

5.1.1 Setting the Directions of the DIOs

The DIOs can be individually configured as inputs and outputs using the function **vAHI_DioSetDirection()** - by default, they are all inputs. If a DIO is shared with an on-chip peripheral and is being used by this peripheral when **vAHI_DioSetDirection()** is called, the specified input/output setting for the DIO will not take immediate effect but will take effect once the peripheral has been disabled.

5.1.2 Setting DIO Outputs

The DIOs configured as outputs can then be individually set to on (high) and off (low) using the function **vAHI_DioSetOutput()**. The output states are set in a 32-bit bitmap, where each DIO is represented by a bit (bits 0-20 are used, bits 21-31 are ignored). Note that:

- DIOs configured as inputs will not be affected by this function unless they are later set as outputs via a call to **vAHI_DioSetDirection()** - they will then adopt the output states set in **vAHI_DioSetOutput()**.
- If a shared DIO is in use by an on-chip peripheral when **vAHI_DioSetOutput()** is called, the specified on/off setting for the DIO will not take immediate effect but will take effect once the peripheral has been disabled.

On the JN5148 device, a set of 8 consecutive DIOs can be used to output a byte in parallel - set DIO0-7 or DIO8-15 can be used for this purpose, where bit 0 or 8 is used for the least significant bit of the byte. The DIO set and the output byte can be specified using the function **vAHI_DioSetByte()**. All DIOs in the selected set must have been previously configured as outputs - see Section 5.1.1.

5.1.3 Setting DIO Pull-ups

Each DIO has an associated pull-up resistor. The purpose of the 'pull-up' is to prevent the state of the pin from 'floating' when there is no external load connected to the DIO - that is, when enabled, the pull-up ties the pin to the high (on) state in the absence of an external load (or in the presence a weak external load). The pull-ups for all the DIOs can be enabled/disabled using the function **vAHI_DioSetPullup()** - by default, all pull-ups are enabled. Again, if a shared DIO is in use by an on-chip peripheral when **vAHI_DioSetPullup()** is called, the specified pull-up setting for the DIO will be applied except when it is connected to an external 32-kHz crystal (JN5148 only - see Section 3.1.4).

Note: DIO pull-up settings are maintained through sleep. A power saving can be made by disabling DIO pull-ups (during sleep or normal operation) if they are not required.

5.1.4 Reading the DIOs

The states of the DIOs can be obtained using the function **u32AHI_DioReadInput()**. This function will return the states of all the DIOs, irrespective of whether they have been configured as inputs or outputs, or are in use by peripherals.

On the JN5148 device, a set of 8 consecutive DIOs can be used to input a byte in parallel - set DIO0-7 or DIO8-15 can be used for this purpose, where bit 0 or 8 is used for the least significant bit of the byte. The input byte on a DIO set can be obtained using the function **u8AHI_DioReadByte()**. All DIOs in the set must have been previously configured as inputs - see Section 5.1.1.

5.2 DIO Interrupts and Wake-up

The DIOs configured as inputs can be used to generate System Controller interrupts. These interrupts can be used to wake the microcontroller, if it is sleeping. The Integrated Peripherals API includes a set of 'DIO interrupt' functions and a set of 'DIO wake' functions, but these functions are identical in their effect (as they access the same register bits in hardware). Use of these two function-sets is described in the subsections below.

Caution: *Since the 'DIO interrupt' and 'DIO wake' functions access the same JN51xx register bits, you must ensure that the two sets of functions do not conflict in your application code.*

5.2.1 DIO Interrupts

A change of state on an input DIO can be used to trigger an interrupt.

First, the input signal transition (low-to-high or high-to-low) that will trigger the interrupt should be selected for individual DIOs using the function **vAHI_DioInterruptEdge()** - the default is a low-to-high transition. Interrupts can then be enabled for the relevant DIO pins using the function **vAHI_DioInterruptEnable()**.

The interrupt status of the DIO pins can subsequently be obtained using the function **u32AHI_DioInterruptStatus()** - that is, this function can be used to determine if one of the DIOs caused an interrupt. This function is useful for polling the interrupt status of the DIOs when DIO interrupts are disabled and therefore not generated.

Note: If DIO interrupts are enabled, you should include DIO interrupt handling in the callback function registered via **vAHI_SysCtrlRegisterCallback()**.

5.2.2 DIO Wake-up

The DIOs can be used to wake the microcontroller from Sleep (including Deep Sleep) or Doze mode. Any DIO pin configured as an input can be used for wake-up - a change of state of the DIO will trigger a wake interrupt.

First, the input signal transition (low-to-high or high-to-low) that will trigger the wake interrupt should be selected for individual DIOs using the function **vAHI_DioWakeEdge()** - the default is a low-to-high transition. Wake interrupts can then be enabled for the relevant DIO pins using the function **vAHI_DioWakeEnable()**.

The wake status of the DIO pins can subsequently be obtained using the function **u32AHI_DioWakeStatus()** - that is, this function can be used to determine if one of the DIOs caused a wake-up event. Note that on waking, you must call this function before **u32AHI_Init()**, as the latter function will clear any pending interrupts.

Note: As an alternative to calling the function **u32AHI_DioWakeStatus()**, you can determine the wake interrupt source in the callback function registered via **vAHI_SysCtrlRegisterCallback()**.

6. UARTs

This chapter describes control of the UARTs (Universal Asynchronous Receiver Transmitters) using functions of the Integrated Peripherals API.

The JN51xx microcontrollers each have two 16550-compatible UARTs, denoted UART0 and UART1, which can be independently enabled. These UARTs can be used for the input/output of serial data at a programmable baud-rate of up to 1 Mbps for the JN5139 device and up to 4 Mbps for the JN5148 device.

6.1 UART Signals and Pins

A UART employs the following signals to interface with an external device:

- Transmit Data (TxD) output - connected to RxD on external device
- Receive Data (RxD) input - connected to TxD on external device
- Request-To-Send (RTS) output - connected to CTS on external device
- Clear-To-Send (CTS) input - connected to RTS on external device

The interface can use just two of these signals (RxD and TxD), in which case it is said to operate in 2-wire mode (see Section 6.2.1), or all four signals, in which case it is said to operate in 4-wire mode and implements flow control (see Section 6.2.2).

The pins used for the above signals are shared with the DIOs, as detailed in the table below:

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

Table 1: DIOs Used for UART Signals

On the JN5148 device, the pins normally used by a UART can alternatively be used to connect a JTAG emulator for debugging.

6.2 UART Operation

The transmit and receive paths of a UART each have a 16-byte deep FIFO buffer, which allows multiple-byte serial transfers to be performed with an external device:

- The TxD pin is connected to the Transmit FIFO
- The RxD pin is connected to the Receive FIFO

On the local device, the CPU can write/read data to/from a FIFO one byte at a time. The two paths are independent, so transmission and reception occur independently.

The basic UART set-up is illustrated in Figure 5 below.

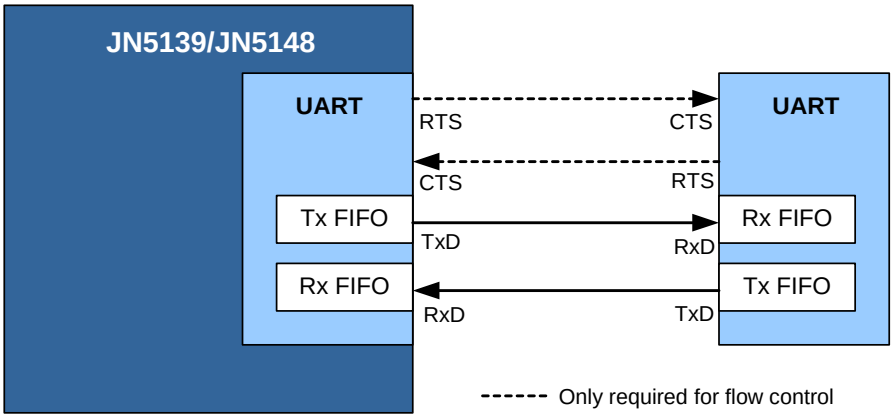


Figure 5: UART Connections

A UART can operate in either 2-wire mode or 4-wire mode, which are introduced in the sub-sections below.

6.2.1 2-wire Mode

In 2-wire mode, the UART only uses signal lines TxD and RxD. Data is transmitted unannounced, at the convenience of the sending device (e.g. when the Transmit FIFO contains some data). Data is also received unannounced and at the convenience of the sending device. This can cause problems and the loss of data - for example, if the receiving device has insufficient space in its Receive FIFO to accept the sent data.

6.2.2 4-wire Mode (with Flow Control)

In 4-wire mode, the UART uses the signal lines TxD, RxD, RTS and CTS. This allows flow control to be implemented, which ensures that sent data can always be accepted. The general principle of flow control is described below.

The RTS and CTS lines are flags that are used to indicate when it is safe to transfer data between the devices. The RTS line on one device is connected to the CTS line on the other device.

The destination device dictates when the source device should send data to it, as follows:

- When the destination device is ready to receive data, it asserts its RTS line to request the source device to send data. This may be when the Receive FIFO fill-level on the destination device falls below a pre-defined level and the FIFO becomes able to receive more data.
- The assertion of the RTS line on the destination device is seen by the source device as the assertion of its CTS line. The source device is then able to send data from its Transmit FIFO.

Flow control operation is illustrated in Figure 6 below.

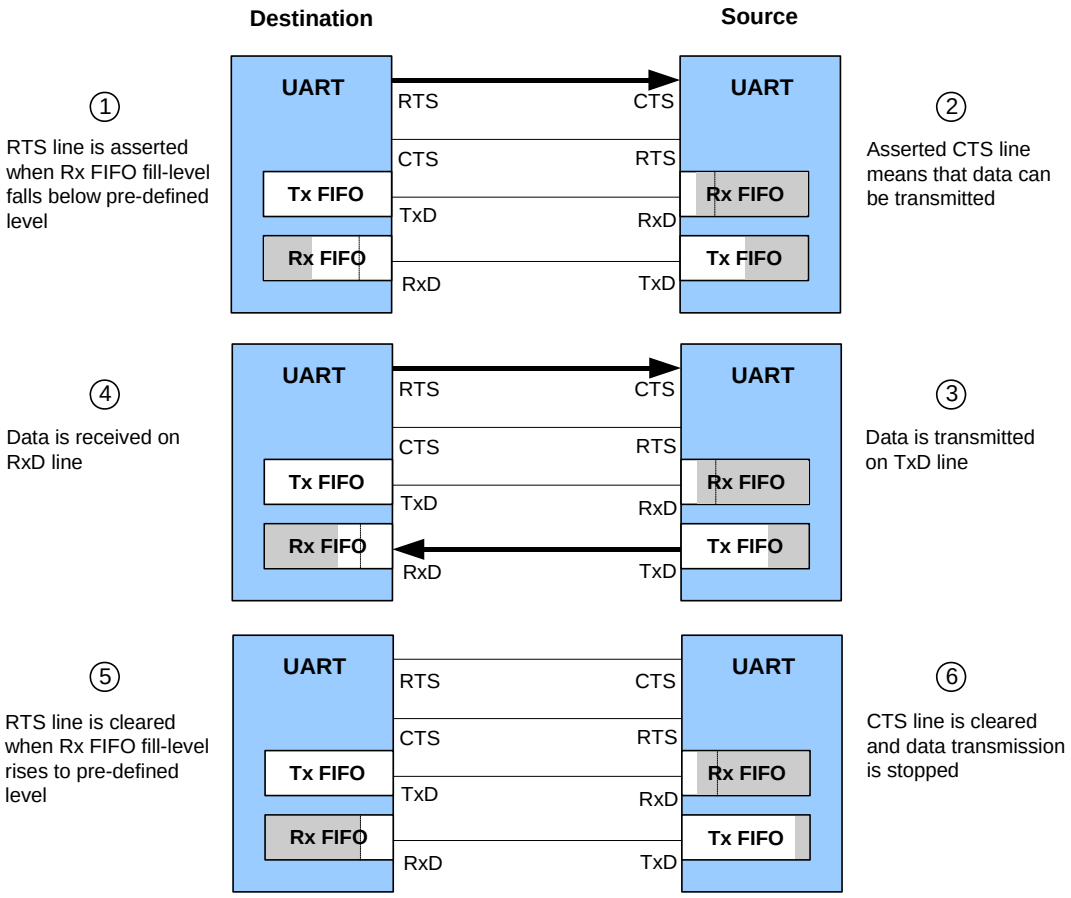


Figure 6: Example of UART Flow Control

The Integrated Peripherals API provides functions for controlling and monitoring the RTS/CTS lines, allowing the application to implement the flow control algorithm manually. In practice, manual flow control can be a burden for a busy CPU, particularly when the UART is operating at a high baud-rate. For this reason, on the JN5148 device, the API provides an Automatic Flow Control option in which the state of the RTS line is controlled directly by the Receive FIFO fill-level on the destination device. The implementations of manual and automatic flow control using the functions of Integrated Peripherals API are described in Section 6.5.

6.3 Configuring the UARTs

This section describes the various aspects of configuring a UART before using it to transfer serial data.

6.3.1 Enabling a UART

A UART is enabled using the function **vAHI_UartEnable()**, which enables the UART in 4-wire mode by default. This must be the first UART function called, unless you wish to use the UART in 2-wire mode (without flow control). In the latter case, you will first need to call **vAHI_UartSetRTSCTS()** in order to release control of the DIOs used for the flow control RTS and CTS lines.

6.3.2 Setting the Baud-rate

The following functions are provided for setting the baud-rate of a UART:

- **vAHI_UartSetBaudRate()**

  This function allows one of the following standard baud-rates to be set: 4800, 9600, 19200, 38400, 76800 or 115200 bps.

- **vAHI_UartSetBaudDivisor()**

  This function allows a 16-bit integer divisor (*Divisor*) to be specified which will be used to derive the baud-rate from a 1-MHz frequency, given by:

$$\frac{1 \times 10^6}{Divisor}$$

- **vAHI_UartSetClocksPerBit() [JN5148 only]**

  This function can be used on the JN5148 device to obtain a more refined baud-rate than can be achieved using **vAHI_UartSetBaudDivisor()** alone. The divisor from the latter function is used in conjunction with an 8-bit integer parameter (*Cpb*) from **vAHI_UartSetClocksPerBit()** to derive a baud-rate from the 16-MHz system clock, given by:

$$\frac{16 \times 10^6}{Divisor \times (Cpb + 1)}$$

  Based on the above formula, the highest recommended baud-rate that can be achieved on the JN5148 device is 4 Mbps (*Divisor*=1, *Cpb*=3).

Note: Either **vAHI_UartSetBaudRate()** or **vAHI_UartSetBaudDivisor()** must be called, but not both. If used, **vAHI_UartSetClocksPerBit()** must be called after **vAHI_UartSetBaudDivisor()**.

6.3.3 Setting Other UART Properties

In addition to setting the baud-rate of a UART, as described in Section 6.3.2, it is also necessary to configure a number of other properties of the UART. These properties are set using the function **vAHI_UartSetControl()** and include the following:

- Parity checks can be optionally applied to the transferred data and the type of parity (odd or even) can be selected.
- The length of a word of data can be set to 5, 6, 7 or 8 bits - this is the number of bits per transmitted 'character' and should normally be set to 8 (a byte).
- The number of stop bits can be set to 1 or 1.5 / 2.
- The initial state of the RTS line can be configured (set or cleared) - this is only implemented if using the UART in the default 4-wire mode (see Section 6.3.1).

6.3.4 Enabling Interrupts

UART interrupts can be generated under a variety of conditions. The interrupts can be enabled and configured using the function **vAHI_UartSetInterrupt()**. The possible interrupt conditions are as follows:

- **Transmit FIFO empty:** The Transmit FIFO has become empty (and therefore requires more data).
- **Receive data available:** The Receive FIFO has filled with data to a pre-defined level, which can be set to 1, 4, 8 or 14 bytes. This interrupt is cleared when the FIFO fill-level falls below the pre-defined level again.
- **Timeout:** This interrupt is enabled when the 'receive data available' interrupt is enabled and is generated if all the following conditions exist:
  - At least one character is in the FIFO.
  - No character has entered the FIFO during a time interval in which at least four characters could potentially have been received.
  - Nothing has been read from the FIFO during a time interval in which at least four characters could potentially have been read.

  A timeout interrupt is cleared and the timer is reset by reading a character from the Receive FIFO.
- **Receive line status:** An error condition has occurred on the RxD line, such as a break indication, framing error, parity error or over-run.
- **Modem status:** A change in the CTS line has been detected (for example, it has been asserted to indicate that the remote device is ready to accept data).

UART interrupts are handled by a callback function which must be registered using the function **vAHI_Uart0RegisterCallback()** or **vAHI_Uart1RegisterCallback()**, depending on the UART (0 or 1). For more information on UART interrupt handling, refer to Section 6.7.

6.4 Transferring Serial Data in 2-wire Mode

In 2-wire mode, a UART only uses signals RxD and TxD, and does not implement flow control. Data transmission and reception are covered separately below.

Note: The default operating mode of a UART is 4-wire mode. In order to use a UART in 2-wire mode, the function **vAHI_UartSetRTSCTS()** must first be called to release control of the DIOs used for flow control. This function must be called before **vAHI_UartEnable()**.

6.4.1 Transmitting Data (2-wire Mode)

Data is transmitted via a UART by simply calling the function **vAHI_UartWriteData()**, which is used by the application to write a single byte of data to the Transmit FIFO. This function should be called multiple times to queue up to 16 data bytes for transmission. Once in the FIFO, a data byte starts to be transmitted as soon as it reaches the head of the FIFO (and provided that the TxD line is idle).

The following methods can be used to prompt the application to call the **vAHI_UartWriteData()** function:

- On the JN5148 device, the function **u8AHI_UartReadTxFifoLevel()** can be called to check the number of characters currently waiting in the Transmit FIFO (more data could then be written to the FIFO, if there is sufficient free space).
- The function **u8AHI_UartReadLineStatus()** can be used to check whether the Transmit FIFO is empty.
- An interrupt can be generated when the Transmit FIFO becomes empty (that is, when the last data byte in the FIFO starts to be transmitted) - this interrupt is enabled using the function **vAHI_UartSetInterrupt()**.
- A timer can be used to schedule periodic transmissions (provided that data is available to be transmitted).

The application can accumulate several bytes of data in its own internal buffer before transferring this data to the Transmit FIFO through repeated calls to **vAHI_UartWriteData()**.

6.4.2 Receiving Data (2-wire Mode)

Data is received in the Receive FIFO (via the RxD line) as and when the source device sends it. The destination application can read a byte of data from the Receive FIFO using the function **u8AHI_UartReadData()**.

The following methods can be used to prompt the application to call the **u8AHI_UartReadData()** function:

- On the JN5148 device, the function **u8AHI_UartReadRxFifoLevel()** can be called to check the number of characters currently in the Receive FIFO.
- The function **u8AHI_UartReadLineStatus()** can be used to check whether the Receive FIFO contains data that can be read (or is empty).
- An interrupt can be generated when the Receive FIFO contains a certain number of data bytes - this interrupt is enabled using the function **vAHI_UartSetInterrupt()**, in which the trigger level for the interrupt must be specified as 1, 4, 8 or 14 bytes.
- A timer can be used to schedule periodic reads of the Receive FIFO. Before each timed read, the presence of data in the FIFO can be checked using either **u8AHI_UartReadLineStatus()** or **u8AHI_UartReadRxFifoLevel()**.

Note: When the 'receive data available' interrupt is enabled (described above), a 'timeout' interrupt is also enabled for the Receive FIFO. For more details of this interrupt, refer to Section 6.3.4.

6.5 Transferring Serial Data in 4-wire Mode

In 4-wire mode, a UART uses the signals RTS and CTS to implement flow control (see Section 6.2.2), as well as RxD and TxD. Flow control can be implemented manually (by the application) or automatically (JN5148 only). The implementation of manual flow control is described below for transmission and reception separately, and then automatic flow control is described.

Note: 4-wire mode is the default operating mode of a UART. Therefore, the UART will automatically have control of the DIOs used for the RTS and CTS lines as soon as **vAHI_UartEnable()** is called.

6.5.1 Transmitting Data (4-wire Mode, Manual Flow Control)

In the flow control protocol, the source device should only transmit data when the destination device is ready to receive (see Section 6.5.2). The readiness of the destination device to accept data is indicated on the source device by its CTS line being asserted. The status of the CTS line can be monitored in either of the following ways:

- The source device can check the status of its CTS line using the function **u8AHI_UartReadModemStatus()**.
- An interrupt can be generated when a change in status of the CTS line occurs - this interrupt is enabled using the function **vAHI_UartSetInterrupt()**.

Once a change in the state of the CTS line (to asserted) has been detected, the function **vAHI_UartWriteData()** can be called to write data to the Transmit FIFO - this function must be called for each byte of data to be transmitted. Once in the FIFO, a data byte starts to be transmitted as soon as it reaches the head of the FIFO (and provided that the TxD line is idle).

Note that before calling **vAHI_UartWriteData()** to write data to the Transmit FIFO, the application may check whether there is already data in the FIFO (left over from a previous transfer) using the function **u8AHI_UartReadTxFifoLevel()** (JN5148 only) or **u8AHI_UartReadLineStatus()**.

The application can accumulate several bytes of data in its own internal buffer before transferring this data to the Transmit FIFO through repeated calls to **vAHI_UartWriteData()**.

The CTS line is de-asserted when the RTS line is de-asserted on the destination device - see Section 6.5.2.

6.5.2 Receiving Data (4-wire Mode, Manual Flow Control)

In the flow control protocol, the destination device should only receive data when it is ready. This is normally when its Receive FIFO has sufficient free space to accept more data. The application can check the fill status of its Receive FIFO using the function **u8AHI_UartReadRxFifoLevel()** (JN5148 only) or **u8AHI_UartReadLineStatus()**.

Once the application on the destination device has decided that it is ready to receive data, it must request the data from the source device by asserting the RTS line (which asserts the CTS line on the source device - see Section 6.5.1). The RTS line can be asserted using the function **vAHI_UartSetRTS()** (JN5148 only) or **vAHI_UartSetControl()**.

The source device may then send data, which is received in the Receive FIFO on the destination device. The received data can be read from the Receive FIFO one byte at a time using the function **u8AHI_UartReadData()**.

The application may subsequently make a decision to stop the transfer from the source device, which is achieved by de-asserting the RTS line using the function **vAHI_UartSetRTS()** (JN5148 only) or **vAHI_UartSetControl()**. This decision is based on the fill-level of the Receive FIFO - when the amount of data in the FIFO reaches a certain level, the application will start to read the data and may also stop the transfer if it cannot read from the FIFO quickly enough to prevent an overflow condition. The current fill-level of the Receive FIFO can be monitored using either of the following mechanisms:

- On the JN5148 device, the function **u8AHI_UartReadRxFifoLevel()** can be called to check the number of data bytes currently in the Receive FIFO.
- A ‘receive data available’ interrupt can be generated when the number of data bytes in the Receive FIFO rises to a certain level - this interrupt is enabled using the function **vAHI_UartSetInterrupt()**, in which the trigger-level for the interrupt must be specified as 1, 4, 8 or 14 bytes.

Note: When the ‘receive data available’ interrupt is enabled (described above), a ‘timeout’ interrupt is also enabled for the Receive FIFO. For more details of this interrupt, refer to Section 6.3.4.

6.5.3 Automatic Flow Control (4-wire Mode) [JN5148 Only]

Flow control can be implemented automatically in UART 4-wire mode on the JN5148 device, rather than manually (as described in Section 6.5.1 and Section 6.5.2). Automatic flow control can be used on the destination device and/or on the source device:

- On the destination device, automatic flow control avoids the need for the application to monitor the Receive FIFO fill-level and to assert/de-assert the RTS line.
- On the source device, automatic flow control avoids the need for the application to monitor the CTS line before transmitting data.

On the JN5148 device, automatic flow control is configured and enabled using the function **vAHI_UartSetAutoFlowCtrl()** which, if used, must be called after enabling the UART and before starting the data transfer.

The **vAHI_UartSetAutoFlowCtrl()** function allows:

- A Receive FIFO trigger-level to be specified on the destination device (as 8, 11, 13 or 15 bytes), so that:
  - The local RTS line is asserted when the fill-level is below the trigger-level, indicating the readiness of the destination device to accept more data.
  - The local RTS line is de-asserted when the fill-level is at or above the trigger-level, indicating that the destination device is not in a position to accept more data.

  Thus, as the destination Receive FIFO fill-level rises and falls (as data is received and read), the local RTS line is automatically manipulated to control the arrival of further data from the source device.
- Automatic monitoring of the CTS line to be enabled on the source device - when this line is asserted, any data in the Transmit FIFO is transmitted automatically.

This function also allows the RTS/CTS signals to be configured as active-high or active-low.

Automatic flow control can be set up between the two devices either for data transfers in only one direction or for data transfers in both directions.

Although much of the data transfer is automatic, the application on the source device must write data into its Transmit FIFO and the application on the destination device must read data from its Receive FIFO. These operations are described below.

Transmitting Data

The sending application must use the function **vAHI_UartWriteData()** to write data to the Transmit FIFO - this function must be called for each byte of data to be transmitted. Once in the FIFO, the data is automatically transmitted (via the TxD line) as soon as the CTS line indicates that the destination device is ready to receive.

Note that before calling **vAHI_UartWriteData()** to write data to the Transmit FIFO, the application may check whether there is already data in the FIFO (left over from a previous transfer) using the function **u8AHI_UartReadTxFifoLevel()** (JN5148 only) or **u8AHI_UartReadLineStatus()**.

The application can accumulate several bytes of data in its own internal buffer before transferring this data to the Transmit FIFO through repeated calls to **vAHI_UartWriteData()**.

Receiving Data

The receiving application must use the function **u8AHI_UartReadData()** to read data from the Receive FIFO, one byte at a time.

The application can decide when to start and stop reading data from the Receive FIFO, based on either of the following mechanisms:

- On the JN5148 device, the function **u8AHI_UartReadRxFifoLevel()** can be called to check the number of characters currently in the Receive FIFO. Thus, the application may decide to start reading data when the FIFO fill-level is at or above a certain threshold. It may decide to stop reading data when the FIFO fill-level is at or below another threshold, or when the FIFO is empty.
- A 'receive data available' interrupt can be generated when the Receive FIFO contains a certain number of data bytes - this interrupt is enabled using the function **vAHI_UartSetInterrupt()**, in which the trigger-level for the interrupt must be specified as 1, 4, 8 or 14 bytes. Thus, the application may decide to start reading data from the Receive FIFO when this interrupt occurs and to stop reading data when all the received bytes have been extracted from the FIFO.

Note: When the 'receive data available' interrupt is enabled (described above), a 'timeout' interrupt is also enabled for the Receive FIFO. For more details of this interrupt, refer to Section 6.3.4.

6.6 Break Condition (JN5148 Only)

During a data transfer from a JN5148 device, if the application on this source device becomes aware of an error, it can convey this error status to the destination device by setting a break condition using the function **vAHI_UartSetBreak()**. When this break condition is issued, the data byte that is currently being transmitted is corrupted and the transmission is stopped.

If a JN5148 device receives a break condition (as the destination device), this results in a 'receive line status' interrupt (E_AHI_UART_INT_RXLINE) being generated on the device, provided that UART interrupts are enabled on this device. UART interrupts are described in Section 6.3.4 and UART interrupt handling in Section 6.7.

The **vAHI_UartSetBreak()** function can also be used to clear the break condition (from the source device). In this case, the transmission will restart in order to transfer the data remaining in the Transmit FIFO.

6.7 UART Interrupt Handling

Interrupts can be employed in a number of ways in controlling UART operation. The various uses of UART interrupts are introduced in Section 6.3.4 and are further covered in the sections on transferring data (Section 6.4 and Section 6.5).

UART interrupts are handled by a user-defined callback function, which must be registered using **vAHI_Uart0RegisterCallback()** or **vAHI_Uart1RegisterCallback()**, depending on the UART (0 or 1). The relevant callback function is automatically invoked when an interrupt of the type E_AHI_DEVICE_UART0 (for UART 0) or E_AHI_DEVICE_UART1 (for UART 1) occurs. For details of the callback function prototype, refer to Appendix A.1.

***Caution:** 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.*

The exact nature of the UART interrupt (from those listed in Section 6.3.4) can then be identified from an enumeration that is passed into the callback function. For details of these enumerations, refer to Appendix B.2.

Note that the handling of UART interrupts differs from the handling of other interrupts in the following ways:

- The exact cause of an interrupt is normally indicated to the callback function by means of a bitmap, but not in the case of a UART interrupt - instead, an enumeration is used to indicate the nature of a UART interrupt. The reported enumeration corresponds to the currently active interrupt condition with the highest priority.
- An interrupt is normally automatically cleared before the callback function is invoked, but the UARTs are the exception to this rule. When generating a 'receive data available' or 'timeout' interrupt, the UART will only clear the interrupt once the data has been read from the Receive FIFO. It is therefore vital that the callback function handles the UART 'receive data available' and 'timeout' interrupts by reading the data from the Receive FIFO before returning.

Note: If the Application Queue API is being used, the above issue with the UART interrupts is handled by this API, so the application does not need to deal with it. For more information on this API, refer to the *Application Queue API Reference Manual (JN-RM-2025)*.

7. Timers

This chapter describes control of the on-chip timers using functions of the Integrated Peripherals API.

The number of timers available depends on the device type:

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

Note: These timers are distinct from the wake timers described in Chapter 8 and tick timer described in Chapter 9.

The timers can operate in a range of modes: Timer, Pulse Width Modulation (PWM), Counter, Capture and Delta-Sigma. These modes are outlined in Section 7.1.

To use a Timer in one of these modes:

1. First refer to Section 7.2 on setting up a timer
2. Then refer to Section 7.3 on operating a timer (you should refer to the sub-section which corresponds to your chosen mode of operation).

For information on Timer interrupts, refer to Section 7.4.

7.1 Modes of Timer Operation

The timers can be operated in the following modes: Timer, Pulse Width Modulation (PWM), Counter, Capture and Delta-Sigma. These modes are summarised in the table below, along with the functions needed for each mode (following a call to **vAHI_TimerEnable()**).

Mode	Description	Functions
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. Timer mode is described further in Section 7.3.1.	**vAHI_TimerConfigureOutputs()** (JN5148) **vAHI_TimerStartSingleShot()** or **vAHI_TimerStartRepeat()**
PWM	As for Timer mode, except the Pulse Width Modulated signal is output on a DIO pin (which depends on the specific timer used - see Section 7.2.1). PWM mode is described further in Section 7.3.1.	**vAHI_TimerConfigureOutputs()** (JN5148) **vAHI_TimerStartSingleShot()** or **vAHI_TimerStartRepeat()**
Counter	The timer is used to count edges on an external input signal, selected as an external clock input. The timer can count just rising edges or both rising and falling edges. Counter mode is described further in Section 7.3.4.	**vAHI_TimerClockSelect()** **vAHI_TimerConfigureInputs()** **vAHI_TimerStartSingleShot()** or **vAHI_TimerStartRepeat()** **u16AHI_TimerReadCount()**
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. Capture mode is described further in Section 7.3.3.	**vAHI_TimerConfigureInputs()** **vAHI_TimerStartCapture()** **vAHI_TimerReadCapture()** or **vAHI_TimerReadCaptureFreeRunning()**
Delta-Sigma	The timer is used as a low-rate DAC. The converted signal is output on a DIO pin (which depends on the specific timer used - see Section 7.2.1) and requires simple filtering to give the analogue signal. Delta-Sigma mode is available in two options, NRZ and RTZ, and is described further in Section 7.3.2.	**vAHI_TimerStartDeltaSigma()**

Table 2: Modes of Timer Operation

7.2 Setting up a Timer

This section describes how to use the Integrated Peripherals API functions to set up a timer before the timer is started (starting and operating a timer are described in Section 7.3).

7.2.1 Selecting DIOs

The timers may 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 and Delta-Sigma output

Table 3: 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

By default, all the DIO pins for an enabled timer are reserved for use by the timer, but these DIOs become available for General Purpose Input/Output (GPIO) when the timer is disabled. Functions are provided that allow the DIO pins associated with an enabled timer to be released for GPIO use. The availability of DIO pins for GPIO use, when the timers are enabled, is summarised in the table below for the JN5139 and JN5148 devices.

Device	DIO Availability
JN5139	When enabled, the timer uses all or none of the assigned DIO pins - the DIOs can be released using the function **vAHI_TimerDIOControl()**. The released DIO pins can then be used for GPIO.
JN5148	When enabled, the timer can use individual DIO pins by releasing unwanted pins using the function **vAHI_TimerFineGrainDIOControl()**. The released DIO pins can then be used for GPIO. Alternatively, the timer can release all of the assigned DIO pins using the function **vAHI_TimerDIOControl()**.

Table 4: DIO Availability During Timer Use

Caution: *The above DIO configuration should be performed before a timer is enabled using* ***vAHI_TimerEnable()****, in order to avoid glitching on the GPIOs during timer operation.*

7.2.2 Enabling a Timer

Before a timer can be started, it must be configured and enabled using the function **vAHI_TimerEnable()**.

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

The **vAHI_TimerEnable()** function contains certain configuration parameters, which are outlined below.

- **Clock Divisor:**

  To obtain the timer frequency, the 16-MHz system clock is divided by a factor of $2^{prescale}$, where *prescale* is a user-configurable integer value in the range 0 to 16 (note that the value 0 leaves the clock frequency unchanged). For example, for a *prescale* value of 3, the 16-MHz system clock is divided by 8 to give a timer frequency of 2 MHz.

- **Interrupts:**

  Each timer can be configured to generate interrupts on either or both of the following conditions:

  - On the rising edge of the timer output (at end of low period)
  - On the falling edge of the timer output (at the end of full timer period)

  Timer interrupts are further described in Section 7.4.

- **External Output:**

  The timer signal can be output externally, but this output must be explicitly enabled. This output is required for Delta-Sigma mode and PWM mode. It is this option which distinguishes between Timer mode (output disabled) and PWM mode (output enabled). The DIO pin on which the timer signal is output depends on the device type:

  - For Timer 0, DIO10 is used
  - For Timer 1, DIO13 is used
  - For Timer 2 (JN5148 only), DIO11 is used

Once a timer has been enabled using **vAHI_TimerEnable()**, an external clock input can be selected (if required - see Section 7.2.3) and then the timer can be started in the desired mode using the relevant start function (see Section 7.3.1 to Section 7.3.4).

Note: An enabled timer can be disabled using the function **vAHI_TimerDisable()**. This stops the timer (if running) and powers down the timer block - this is useful to reduce power consumption when the timer is not needed. The application must not attempt to access a disabled timer, otherwise an exception may occur.

7.2.3 Selecting the Clock

Each timer requires a source clock, which is by default the internal 16-MHz clock. This source clock is divided down to produce the timer's clock. The division factor is specified when the timer is enabled using **vAHI_TimerEnable()** - see Section 7.2.2.

When operating in Counter mode on the JN5148 device (see Section 7.3.4), an external clock is monitored by the timer. This signal is input on a DIO pin that is dependent on the timer - DIO8 for Timer 0, DIO11 for Timer 1 (Counter mode is not supported on Timer 2). This external input for Counter mode must be selected using the function **vAHI_TimerClockSelect()**, which must be called after **vAHI_TimerEnable()**.

7.3 Starting and Operating a Timer

This section describes how to use the Integrated Peripherals API functions to start and operate a timer that has been set up as described in Section 7.2. A timer can be started in the following modes:

- Timer or PWM mode - see Section 7.3.1
- Delta-Sigma mode - see Section 7.3.2
- Capture mode - see Section 7.3.3
- Counter mode (JN5148 only) - see Section 7.3.4

7.3.1 Timer and PWM Modes

Timer mode allows a timer to produce a rectangular waveform of a specified period, where this waveform starts low and then goes high after a specified time. These times are specified when the timer is started (see below), in terms of the following parameters:

- **Time to rise (*u16Hi*):** This is the number of clock cycles between starting the timer and the (first) low-to-high transition. An interrupt can be generated at this transition.
- **Time to fall (*u16Lo*):** This is the number of clock cycles between starting the timer and the (first) high-to-low transition (effectively the period of one pulse cycle). An interrupt can be generated at this transition.

These times and the timer signal are illustrated below in Figure 7.

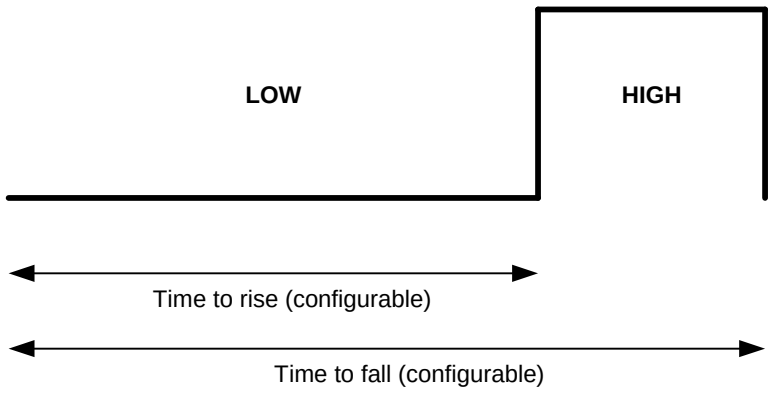


Figure 7: Timer Mode Signal

Within Timer mode, there are two sub-modes and the timer is started in these modes using different functions:

- **Single-shot mode:** The timer produces a single pulse cycle (as depicted in Figure 7) and then stops. The timer can be started in this mode using **vAHI_TimerStartSingleShot()**.
- **Repeat mode:** The timer produces a train of pulses (where the repetition rate is determined by the configured 'time to fall' period - see above). The timer can be started in this mode using **vAHI_TimerStartRepeat()**.

Once started, the timer can be stopped using the function **vAHI_TimerStop()**.

PWM (Pulse Width Modulation) mode is identical to Timer mode except the produced waveform is output on a DIO pin - DIO10 for Timer 0, DIO13 for Timer 1 and DIO11 for Timer 2 (JN5148 only). This output can be enabled in **vAHI_TimerEnable()**. The output can also be inverted using the function **vAHI_TimerConfigureOutputs()** for JN5148.

7.3.2 Delta-Sigma Mode (NRZ and RTZ)

Delta-Sigma mode allows a timer to be used as a simple low-rate DAC. This requires the timer output to be enabled in **vAHI_TimerEnable()**. The output pin is DIO10 for Timer 0, DIO13 for Timer 1 and DIO11 for Timer 2 (JN5148 only). An RC (Resistor-Capacitor) circuit must be inserted between this pin and Ground (see Figure 8).

A timer is started in Delta-Sigma mode using **vAHI_TimerStartDeltaSigma()**. The value to be converted is digitally encoded by the timer as a pseudo-random waveform in which:

- the total number of clock cycles that make up one period of the waveform is fixed (at 2^{16} for NRZ and at 2^{17} for RTZ - see below)
- the number of high clock cycles during one period is set to a number which is proportional to the value to be converted
- the high clock cycles are distributed randomly throughout a complete period

Thus, the capacitor will charge in proportion to the specified value such that, at the end of the period, the voltage produced is an analogue representation of the digital value. The output voltage requires calibration - for example, you could determine the maximum possible voltage by measuring the voltage across the capacitor after a conversion with the high period set to the whole pulse period (less one clock cycle).

Two Delta-Sigma mode options are available, NRZ and RTZ:

- **NRZ (Non Return-to-Zero):** Delta-Sigma NRZ mode uses the 16-MHz system clock and the period of the waveform is fixed at 2^{16} clock cycles. The NRZ option means that clock cycles are implemented without gaps between them (see RTZ option below). You must define the number of clock cycles spent in the high state during the pulse cycle such that this high period is proportional to the value to be converted. This number is set when the timer is started using the function **vAHI_TimerStartDeltaSigma()**. For example, if you wish to convert values in the range 0-100 then 2^{16} clock cycles would correspond to 100, and to convert the value 25 you must set the number of high clock cycles to 2^{14} (a quarter of the pulse cycle). For an illustration, refer to Figure 8.
- **RTZ (Return-to-Zero):** Delta-Sigma RTZ mode is similar to the NRZ option, described above, except that after every clock cycle, a blank (low) clock cycle is inserted. Thus, each pulse cycle takes twice as many clock cycles - that is, 2^{17}. Note that this does not affect the required number of high clock cycles to represent the digital value being converted. This mode doubles the conversion period but improves linearity if the rise and fall times of the outputs are different from each other.

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

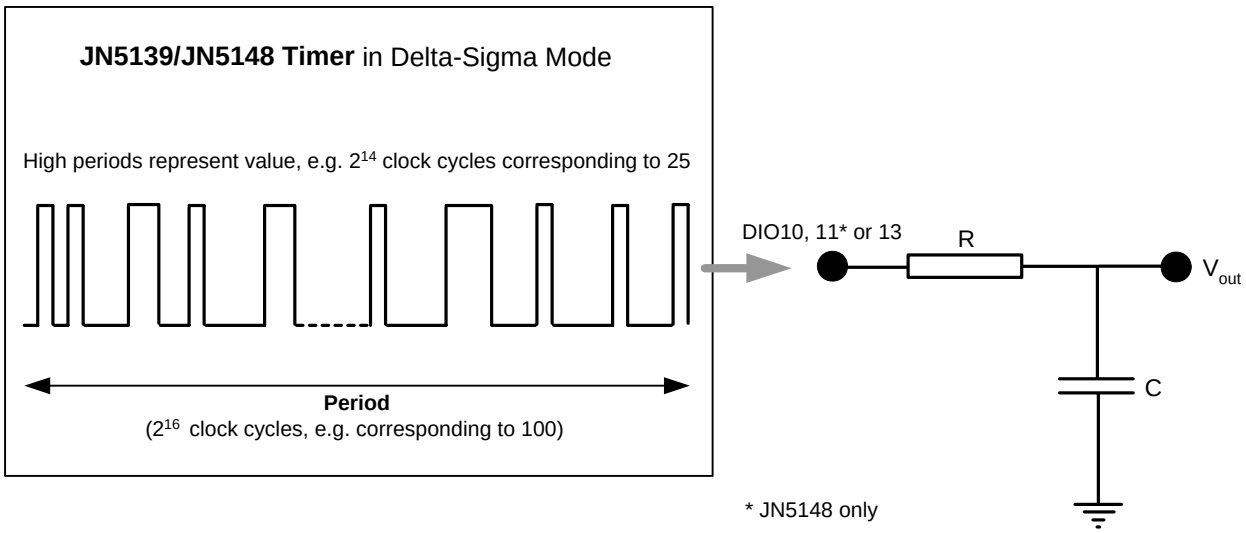


Figure 8: Delta-Sigma NRZ Mode Operation

7.3.3 Capture Mode

In Capture mode, Timer 0 or Timer 1 can be used to measure the pulse width of an external input (Capture mode is not available on Timer 2 of the JN5148 device). The external signal must be provided on the DIO9 pin (Timer 0) or DIO12 pin (Timer 1). The timer measures the number of clock cycles in the input signal from the start of capture to the next low-to-high transition and also to next the high-to-low transition. The number of clock cycles in the last pulse is then the difference between these measured values (see Figure 9). The pulse width in units of time is then given by:

Pulse width (in units of time) = Number of clock cycles in pulse X Clock cycle period

A timer is started in Capture mode using the function **vAHI_TimerStartCapture()**. The timer can be stopped and the most recent measurements obtained using the function **vAHI_TimerReadCapture()**. These measurements can alternatively be obtained without stopping the timer by calling **vAHI_TimerReadCaptureFreeRunning()**.

Note: Only the measurements for the last low-to-high and high-to-low transitions are stored, and then returned when the above 'read capture' functions are called. Therefore, it is important not to call these functions during a pulse, as in this case the measurements will not give sensible results. To ensure that you obtain the capture results after a pulse has completed, you should enable interrupts on the falling edge when the timer is configured using **vAHI_TimerEnable()**.

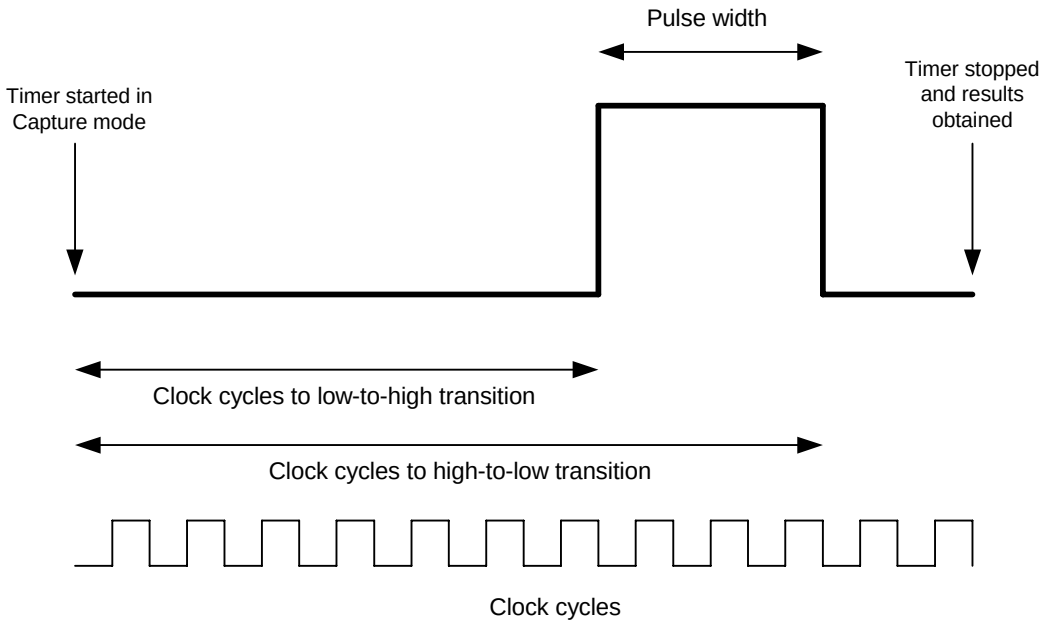


Figure 9: Capture Mode Operation

On the JN5148 device, the input signal for Capture mode can be inverted. This option is configured using the function **vAHI_TimerConfigureInputs()** and allows the low-pulse width (instead of the high-pulse width) of the input signal to be measured.

7.3.4 Counter Mode (JN5148 Only)

Counter mode is available on Timer 0 and Timer 1 of the JN5148 device to count edges on an external clock signal (Counter mode is not available on Timer 2). The input signal must be provided on DIO9 (Timer 0) or DIO12 (Timer 1). Counter mode is enabled by selecting an external clock input in a call to **vAHI_TimerClockSelect()**.

The timer can count rising edges only or both rising and falling edges. This must be configured using the function **vAHI_TimerConfigureInputs()**. Edges must be at least 100 ns apart, i.e. pulses must be wider than 100 ns.

Like Timer/PWM mode, the timer can then be started in one of two sub-modes:

- **Single-shot mode:** The timer can be started in this mode using the function **vAHI_TimerStartSingleShot()** and will stop at a specified count value (*u16Lo*).
- **Repeat mode:** The timer can be started in this mode using the function **vAHI_TimerStartRepeat()**. The timer operates continuously and the counter resets to zero each time the specified count value (*u16Lo*) is reached.

The above start functions each allow two counts to be specified at which interrupts will be generated (timer interrupts must also have been enabled in **vAHI_TimerEnable()**).

The current count of a running timer can be obtained at any time using the function **u16AHI_TimerReadCount()**. The timer can be stopped using **vAHI_TimerStop()**.

7.4 Timer Interrupts

A timer can be configured in **vAHI_TimerEnable()** to generate interrupts on either or both of the following conditions:

- On the rising edge of the timer output (at end of low period)
- On the falling edge of the timer output (at the end of full timer period)

The handling of timer interrupts must be incorporated in a user-defined callback function for the particular timer. These callback functions are registered using dedicated registration functions for the individual timers:

- **vAHI_Timer0RegisterCallback()** for Timer 0
- **vAHI_Timer1RegisterCallback()** for Timer 1
- **vAHI_Timer2RegisterCallback()** for Timer 2 (JN5148 only)

The relevant callback function is automatically invoked when an interrupt of the type E_AHI_DEVICE_TIMER0, E_AHI_DEVICE_TIMER1 or E_AHI_DEVICE_TIMER2 occurs. The exact nature of the interrupt (from the two conditions listed above) can then be identified from a bitmap that is passed into the function. Note that the interrupt will be automatically cleared before the callback function is invoked.

Note: The callback function prototype is detailed in Appendix A.1. The interrupt source information is provided in Appendix B.

Caution: *A 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.*

8. Wake Timers

This chapter describes control of the on-chip wake timers using functions of the Integrated Peripherals API.

The JN51xx microcontrollers include two wake timers, denoted Wake Timer 0 and Wake Timer 1. These are 32-bit timers on the JN5139 device and 35-bit timers on the JN5148 device. The wake timers are based on the internal 32-kHz clock and can run while the device is in sleep mode (and while the CPU is running). They are generally used to time the sleep duration and wake the device at the end of the sleep period. A wake timer counts down from a programmed value and wakes the device when the count reaches zero by generating an interrupt or wake-up event.

8.1 Using a Wake Timer

This section describes how to use the Integrated Peripherals API functions to operate a wake timer.

8.1.1 Enabling and Starting a Wake Timer

A wake timer is enabled using the function **vAHI_WakeTimerEnable()**. This function allows the interrupt to be enabled/disabled that is generated when the counter reaches zero. Note that wake timer interrupts are handled by the callback function registered using the function **vAHI_SysCtrlRegisterCallback()** - see Section 3.5.

The wake timer can then be started using one of the following functions:

- **vAHI_WakeTimerStart()** is used to start a 32-bit wake timer on the JN5139 device.
- **vAHI_WakeTimerStartLarge()** is used to start a 35-bit wake timer on the JN5148 device.

This function takes as a parameter the starting value for the countdown - this value must be specified in 32-kHz clock periods (thus, 32 corresponds to 1 millisecond).

On reaching zero, the timer ‘fires’, rolls over (to 0xFFFFFFFF on JN5139 or 0x7FFFFFFF on JN5148) and continues to count down. If enabled, the wake timer interrupt is generated on reaching zero.

Note: The 32-kHz internal clock, which drives the wake timers, may be running up to 30% fast or slow. For accurate timings, you are advised to first calibrate the clock and adjust the specified count value accordingly, as described in Section 8.2.

8.1.2 Stopping a Wake Timer

A wake timer can be stopped at any time using the function **vAHI_WakeTimerStop()**. The counter will then remain at the value at which it was stopped and will not generate an interrupt.

8.1.3 Reading a Wake Timer

The current count of a wake timer can be obtained using one of the following functions:

- **u32AHI_WakeTimerRead()** is used to read a 32-bit wake timer on the JN5139 device.
- **u64AHI_WakeTimerReadLarge()** is used to read a 35-bit wake timer on the JN5148 device.

These functions do not stop the wake timer.

8.1.4 Obtaining Wake Timer Status

The states of the wake timers can be obtained using the following functions:

- **u8AHI_WakeTimerStatus()** can be used to find out which wake timers are currently running.
- **u8AHI_WakeTimerFiredStatus()** can be used to find out which wake timers have fired (passed zero). The 'fired' status of a wake timer is also cleared by this function.

Note: If using **u8AHI_WakeTimerFiredStatus()** to check whether a wake timer caused a wake-up event, you must call this function before **u32AHI_Init()**.

8.2 Clock Calibration

The wake timers are driven by the microcontroller's internal 32-kHz clock. However, this clock may run up to 30% fast or slow, depending on temperature, supply voltage and manufacturing tolerance. For cases in which accurate timing is required, a self-calibration facility is provided to time the 32-kHz clock against the chip's more accurate 16-MHz clock. This test is performed using Wake Timer 0. The result of this calibration allows you to calculate the required number of 32-kHz clock cycles to achieve the desired timer duration when starting a wake timer with the function **vAHI_WakeTimerStart()** or **vAHI_WakeTimerStartLarge()**.

The calibration is performed using the function **u32AHI_WakeTimerCalibrate()**, as described below.

1. Wake Timer 0 must be disabled (using **vAHI_WakeTimerStop()**, if required).
2. The status of both wake timers (0 and 1) must be cleared by calling the function **u8AHI_WakeTimerFiredStatus()**.
3. The calibration is started using **u32AHI_WakeTimerCalibrate()**.

   This causes Wake Timer 0 to start counting down 20 clock periods of the internal 32-kHz clock. At the same time, a reference counter starts counting up from zero using the 16-MHz clock.
4. When the wake timer reaches zero, **u32AHI_WakeTimerCalibrate()** returns the number of 16-MHz clock cycles registered by the reference counter. Let this value be *n*.
   - If the clock is running at 32 kHz, $n = 10000$
   - If the clock is running slower than 32 kHz, $n > 10000$
   - If the clock is running faster than 32 kHz, $n < 10000$
5. You can then calculate the required number of 32-kHz clock periods (for **vAHI_WakeTimerStart()** or **vAHI_WakeTimerStartLarge()**) to achieve the desired timer duration. If *T* is the required duration in seconds, the appropriate number of 32-kHz clock periods, *N*, is given by:

$$N = \left(\frac{10000}{n}\right) \times 32000 \times T$$

For example, if a value of 9000 is obtained for *n*, this means that the 32-kHz clock is running fast. Therefore, to achieve a 2-second timer duration, instead of requiring 64000 clock periods, you will need (10000/9000) x 32000 x 2 clock periods; that is, 71111 (rounded down).

Tip: To ensure that the device wakes in time for a scheduled event, it is better to under-estimate the required number of 32-kHz clock periods than to over-estimate them.

9. Tick Timer

This chapter describes control of the Tick Timer using functions of the Integrated Peripherals API.

The Tick Timer is a hardware timer derived from the 16-MHz system clock. It can be used to implement:

- timing interrupts to software
- 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 that on the JN5139 device, the Tick Timer stops when the CPU enters Doze mode and therefore cannot be used to bring the CPU out of Doze mode.

9.1 Tick Timer Operation

The Tick Timer counts upwards until the count matches a pre-defined reference value (the starting value can be specified). The timer can be operated in one of three modes, which determine 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)

An interrupt can also be enabled which is generated on reaching the reference count.

9.2 Using the Tick Timer

This section describes how to use the Integrated Peripherals API functions to set up and run the Tick Timer.

9.2.1 Setting Up the Tick Timer

On device power-up/reset, the Tick Timer is disabled. However, before setting up the Tick Timer, you are advised to call the function **vAHI_TickTimerConfigure()** and specify the disable option. The starting count and reference count can then be set as follows:

1. The starting count is set (in the range 0 to 0xFFFFFFFF) using the function **vAHI_TickTimerWrite()**. Note that if this function is called while the timer is enabled, the timer will immediately start counting from the specified value.
2. The reference count is set (in the range 0 to 0x0FFFFFFF) using the function **vAHI_TickTimerInterval()**.

9.2.2 Running the Tick Timer

Once the timer has been set up (as described in Section 9.2.1), it can be started by calling the function **vAHI_TickTimerConfigure()** again but, this time, specifying one of the three operational modes listed in Section 9.1.

The current count of the Tick Timer can be obtained at any time by calling the function **u32AHI_TickTimerRead()**.

Note that if the Tick Timer is started 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()**.

9.3 Tick Timer Interrupts

An interrupt can be enabled that will be generated when the Tick Timer reaches its reference count. This interrupt is enabled using the function **vAHI_TickTimerIntEnable()**.

The Tick Timer interrupt is handled by a user-defined callback function which is registered using one of the following functions, depending on the chip type:

- **vAHI_TickTimerRegisterCallback()** for JN5148
- **vAHI_TickTimerInit()** for JN5139

The registered callback function is automatically invoked when an interrupt of the type E_AHI_DEVICE_TICK_TIMER occurs. For details of the callback function prototype, refer to Appendix A.1.

Caution: *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.*

The following functions are also provided to deal with the status of the Tick Timer interrupt:

- **bAHI_TickTimerIntStatus()** obtains the current interrupt status of the Tick Timer.
- **vAHI_TickTimerIntPendClr()** clears a pending Tick Timer interrupt.

10. Watchdog Timer (JN5148 Only)

This chapter describes control of the Watchdog Timer on the JN5148 device using functions of the Integrated Peripherals API.

The Watchdog Timer is provided to allow the JN5148 device to recover from software lock-ups. Note that a watchdog can also be implemented (on all JN51xx devices) using the Tick Timer, described in Chapter 9.

10.1 Watchdog Operation

The Watchdog Timer is derived from the 32-kHz RC oscillator and implements a timeout period. On reaching this timeout period, the JN5148 device is automatically reset. Therefore, to avoid a chip reset, the application must regularly reset the Watchdog Timer (to the start of the timeout period) 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.

Note that 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).

Note: Following a power-up, reset or wake-up from sleep, the Watchdog Timer is enabled with the maximum possible timeout period of 16392 ms (regardless of its state before any sleep or reset).

10.2 Using the Watchdog Timer

This section describes how to use the Integrated Peripherals API functions to start and reset the Watchdog Timer.

10.2.1 Starting the Timer

The Watchdog Timer is started by default on the JN5148 device. It is started with the maximum possible timeout of 16392 ms.

- If the Watchdog Timer is required with a shorter timeout period, the timer must be restarted with the desired period. To do this, first call the function **vAHI_WatchdogRestart()** to restart the timer from the beginning of the timeout period and then call the function **vAHI_WatchdogStart()** to specify the new timeout period (see below).
- If the Watchdog Timer is not required in the application, call the function **vAHI_WatchdogStop()** at the start of your code to stop the timer.

In the function **vAHI_WatchdogStart()**, the timeout period must be specified via an index, *Prescale* (in the range 0 to 12), which the function uses to calculate the timeout period, in milliseconds, according to the following formulae:

Timeout Period = 8 ms if *Prescale* = 0

Timeout Period = $[2^{(Prescale - 1)} + 1] \times 8$ ms if $1 \leq Prescale \leq 12$

This gives timeout periods in the range 8 to 16392 ms.

Note that 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: If called while the Watchdog Timer is in a stopped state, **vAHI_WatchdogStart()** will start the timer with the specified timeout period. If this function is called while the timer is running, the timer will continue to run but with the newly specified timeout period.

The current count of a running Watchdog Timer can be obtained using the function **u16AHI_WatchdogReadValue()**.

10.2.2 Resetting the Timer

A running Watchdog Timer should be reset by the application before the pre-set timeout period is reached. This is done using the function **vAHI_WatchdogRestart()**, which restarts the timer from the beginning of the timeout period. When applying this reset, the application should take into account the fact that the true timeout period may be up to 30% shorter than the calculated timeout period (see Section 10.2.1).

If the application fails to prevent a Watchdog timeout, the chip will be automatically reset. The function **bAHI_WatchdogResetEvent()** can be used following a chip reset to find out whether the last hardware reset was caused by a Watchdog Timer expiry event.

Note that it is also possible to stop the Watchdog Timer and freeze its count by using the function **vAHI_WatchdogStop()**.

11. Pulse Counters (JN5148 Only)

This chapter describes control of the pulse counters on the JN5148 device using functions of the Integrated Peripherals API.

Two pulse counters are provided on the JN5148 device, Pulse Counter 0 and Pulse Counter 1. A pulse counter detects and counts pulses in an external signal that is input on an associated DIO pin.

11.1 Pulse Counter Operation

The two pulse counters on the JN5148 device, Pulse Counter 0 and Pulse Counter 1, are each 16-bit counters which receive their input signals on pins DIO1 and DIO8, respectively. The two counters can be combined together to form a single 32-bit counter, if desired, in which case the input signal is taken from the DIO1 pin.

The pulse counters can operate in all power modes of the JN5148 device, including sleep, and with input signals of up to 100 kHz. An increment of the counter can be configured to occur on a rising or falling edge of the relevant input. Each pulse counter has an associated user-defined reference value. An interrupt (or wake-up event, if asleep) can be generated when the counter passes its pre-configured reference value - that is, when the count reaches *(reference value + 1)*. The counters do not saturate at their maximum count values, but wrap around to zero.

Note: Pulse counter interrupts are handled by the callback function for the System Controller interrupts, registered using **vAHI_SysCtrlRegisterCallback()** - see Section 11.3.

Debounce

The input pulses can be debounced using the 32-kHz clock, to avoid false counts on slow or noisy edges. The debounce feature requires a number of identical consecutive input samples (2, 4 or 8) before a change in the input signal is recognised. Depending on the debounce setting, a pulse counter can work with input signals up to the following frequencies:

- 100 kHz, if debounce disabled
- 3.7 kHz, if debounce enabled to operate with 2 consecutive samples
- 2.2 kHz, if debounce enabled to operate with 4 consecutive samples
- 1.2 kHz, if debounce enabled to operate with 8 consecutive samples

The required debounce setting is selected when the pulse counter is configured, as described in Section 11.2.1.

When using debounce, the 32-kHz clock must be active - therefore, for minimum sleep current, the debounce feature should not be used.

11.2 Using a Pulse Counter

This section describes how to use the Integrated Peripherals API functions to configure, start/stop and monitor a pulse counter.

11.2.1 Configuring a Pulse Counter

A pulse counter must first be configured using the **bAHI_PulseCounterConfigure()** function. This function call must specify:

- if the two 16-bit pulse counters are to be combined into a single 32-bit pulse counter
- if the pulse count is to be incremented on the rising edge or falling edge of a pulse in the input signal
- if the debounce feature is to be enabled and, if so, the number of consecutive samples (2, 4 or 8) with which it will operate (see Section 11.1)
- if an interrupt is to be enabled which is generated when the pulse count passes the reference value (see below)

When a pulse counter is selected using this function, the input signal will automatically be taken from the relevant pin: DIO1 for Pulse Counter 0, DIO8 for Pulse Counter 1 and DIO1 for the combined pulse counter.

The configuration of the pulse counter is completed by calling the function **bAHI_SetPulseCounterRef()** in order to set the reference count. Note that the pulse counter will continue to count beyond the specified reference value, but will wrap around to zero on reaching the maximum possible count value.

11.2.2 Starting and Stopping a Pulse Counter

A configured pulse counter is started using the function **bAHI_StartPulseCounter()**. Note that the count may increment by one when this function is called (even though no pulse has been detected).

The pulse counter will continue to count until stopped using the function **bAHI_StopPulseCounter()**, at which point the count will be frozen. The count can then be cleared to zero using one of the following functions:

- **bAHI_Clear16BitPulseCounter()** for Pulse Counter 0 or 1
- **bAHI_Clear32BitPulseCounter()** for the combined pulse counter

11.2.3 Monitoring a Pulse Counter

The application can detect whether a running pulse counter has reached its reference count in either of the following ways:

- An interrupt can be enabled which is triggered when the reference count is passed (see Section 11.3).
- The application can use the function **u32AHI_PulseCounterStatus()** to poll the pulse counters - this function returns a bitmap which includes all running pulse counters and indicates whether each counter has reached its reference value.

Functions are also provided that allow the current count of a pulse counter to be read without stopping the pulse counter or clearing its count. The required function depends on the pulse counter:

- **bAHI_Read16BitCounter()** for Pulse Counter 0 or 1
- **bAHI_Read32BitCounter()** for the combined pulse counter

When a pulse counter reaches its reference count, it continues counting beyond this value. If required, a new reference count can then be set (while the counter is running) using the function **bAHI_SetPulseCounterRef()**.

11.3 Pulse Counter Interrupts

A pulse counter can optionally generate an interrupt when its count passes the pre-set reference value - that is, when the count reaches *(reference value + 1)*. This interrupt can be enabled as part of the call to the function **bAHI_PulseCounterConfigure()**.

Note: A pulse counter continues to run during sleep. A pulse counter interrupt can be used to wake the JN5148 device from sleep.

The pulse counter interrupt is handled as a System Controller interrupt and must therefore be incorporated in the user-defined callback function registered using the function **vAHI_SysCtrlRegisterCallback()** - see Section 3.5.

The registered callback function is automatically invoked when an interrupt of the type E_AHI_DEVICE_SYSCTRL occurs. If the source of the interrupt is Pulse Counter 0 or Pulse Counter 1, this will be indicated in the bitmap that is passed into the callback function (if the combined pulse counter is in use, this counter will be shown as Pulse Counter 0 for the purpose of interrupts). Note that the interrupt will be automatically cleared before the callback function is invoked.

Once a pulse counter interrupt has occurred, the pulse counter will continue to count beyond its reference value. If required, a new reference count can then be set (while the counter is running) using the function **bAHI_SetPulseCounterRef()**.

12. Serial Interface (SI)

This chapter describes control of the 2-wire Serial Interface (SI) using functions of the Integrated Peripherals API.

The JN51xx microcontrollers include an industry-standard 2-wire synchronous Serial Interface that provides a simple and efficient method of data exchange between devices. The Serial Interface is similar to an I^2C interface and comprises two lines:

- Serial data line
- Serial clock line

The SI peripheral on a JN51xx device can act as a master or a slave of the Serial Interface bus, depending on the device:

- An SI master is a feature of the JN51xx microcontrollers - see Section 12.1.
- An SI slave is provided only on the JN5148 device - see Section 12.2.

Tip: The protocol used by the Serial Interface is detailed in the I^2C Specification (available from www.nxp.com).

12.1 SI Master

The SI master can implement communication in either direction with a slave device on the Serial Interface bus. This section describes how to implement a data transfer.

Note: The Serial Interface 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 on to resolve conflicts caused by competing masters attempting to take control of the Serial Interface bus. If a master loses arbitration, it must wait and try again later.

12.1.1 Enabling the SI Master

The SI master has its own set of functions in the Integrated Peripherals API (and, for JN5148, the SI slave has a separate set of functions). Before using any of the SI master functions, the SI peripheral must be enabled using the function **vAHI_SiConfigure()** for JN5139 or **vAHI_SiMasterConfigure()** for JN5148.

When enabled, this interface uses the DIO14 pin as a clock line and the DIO15 pin as a bi-directional data line. As a bus master, the microcontroller provides the clock (on the clock line) for synchronous data transfers (on the data line), where the clock is scaled from the on-chip 16-MHz clock. The clock scaling factor, *PreScaler*, is specified when the interface is enabled - the final operating frequency of the interface is given by:

Operating frequency = 16/[(*PreScaler* + 1) x 5] MHz

The SI enable functions also allow SI interrupts (of the type E_AHI_DEVICE_SI) to be enabled, which are handled by the user-defined callback function registered using the function **vAHI_SiRegisterCallback()**. For details of the callback function prototype, refer to Appendix A.1.

Caution: *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.*

For JN5148, **vAHI_SiMasterConfigure()** also allows a pulse suppression filter to be enabled, which suppresses any spurious pulses (high or low) with a pulse width less than 62.5 ns on the clock and data lines. Also note that a JN5148 SI master enabled using this function can later be disabled using **vAHI_SiMasterDisable()**.

12.1.2 Writing Data to SI Slave

The procedure below describes how the SI master can write data to an SI slave which has a 7-bit or 10-bit address. It is assumed that the SI master has been enabled as described in Section 12.1.1. The data can comprise one or more bytes.

Step 1 **Take control of SI bus and write slave address to bus**

The SI master must first take control of the SI bus and transmit the address of the target slave for the data transfer. The required method is different for 7-bit and 10-bit slave addresses, as outlined below:

For 7-bit slave address:

a) Call the function **vAHI_SiMasterWriteSlaveAddr()** to specify the 7-bit slave address. Also specify through this function that a write operation will be performed on the slave. This function will put the specified slave address in the SI master's buffer, but will not transmit it on the SI bus.

b) Call the function **bAHI_SiMasterSetCmdReg()** to issue Start and Write commands, in order to take control of the SI bus and transmit the slave address specified above.

c) Wait for an indication of success (slave address sent and target slave responded) by polling or waiting for an interrupt - for details of this stage, refer to Section 12.1.4.

For 10-bit slave address:

a) Call the function **vAHI_SiMasterWriteSlaveAddr()** to indicate that 10-bit slave addressing will be used and to specify the two most significant bits of the relevant slave address (when specified, these bits must be logically ORed with 0x78). Also specify through this function that a write operation will be performed on the slave. This function will put the specified information in the SI master's buffer, but will not transmit it on the SI bus.

b) Call the function **bAHI_SiMasterSetCmdReg()** to issue Start and Write commands, in order to take control of the SI bus and transmit the slave address information specified above.

c) Wait for an indication of success (slave address information sent and at least one matching slave responded) by polling or waiting for an interrupt - for details of this stage, refer to Section 12.1.4.

d) Call the function **vAHI_SiMasterWriteData8()** to specify the eight remaining bits of the slave address. This function will put the specified information in the SI master's buffer, but will not transmit it on the SI bus.

e) Call the function **bAHI_SiMasterSetCmdReg()** to issue a Write command, in order to transmit the slave address information specified above.

f) Wait for an indication of success (slave address information sent and target slave responded) by polling or waiting for an interrupt - for details of this stage, refer to Section 12.1.4.

Step 2 **Send data byte to slave**

If only one data byte or the final data byte is to be sent to the slave then go directly to Step 3, otherwise follow the instructions below:

a) Call the function **vAHI_SiMasterWriteData8()** to specify the data byte to be sent. This function will put the specified data in the SI master's buffer, but will not transmit it on the SI bus.

b) Call the function **bAHI_SiMasterSetCmdReg()** to issue a Write command, in order to transmit the data byte specified above.

c) Wait for an indication of success (data byte sent and target slave responded) by polling or waiting for an interrupt - for details of this stage, refer to Section 12.1.4.

Repeat the above instructions (Step 2a-c) for all subsequent data bytes except the final byte to be sent (which is covered in Step 3).

Step 3 **Send final data byte to slave**

Send the final (or only) data byte to the slave as follows:

a) Call the function **vAHI_SiMasterWriteData8()** to specify the data byte to be sent. This function will put the specified data in the SI master's buffer, but will not transmit it on the SI bus.

b) Call the function **bAHI_SiMasterSetCmdReg()** to issue Write and Stop commands, in order to transmit the data byte specified above and release control of the SI bus.

c) Wait for an indication of success (data byte sent and target slave responded) by polling or waiting for an interrupt - for details of this stage, refer to Section 12.1.4.

12.1.3 Reading Data from SI Slave

The procedure below describes how the SI master can read data sent from an SI slave which has a 7-bit or 10-bit address. It is assumed that the SI master has been enabled as described in Section 12.1.1. The data can comprise one or more bytes.

Step 1 **Take control of SI bus and write slave address to bus**

The SI Master must first take control of the SI bus and transmit the address of the slave which is to be the source of the data transfer. The required method is different for 7-bit and 10-bit slave addresses, as outlined below:

For 7-bit slave address:

a) Call the function **vAHI_SiMasterWriteSlaveAddr()** to specify the 7-bit slave address. Also specify through this function that a read operation will be performed on the slave. This function will put the specified slave address in the SI master's buffer, but will not transmit it on the SI bus.

b) Call the function **bAHI_SiMasterSetCmdReg()** to issue Start and Write commands, in order to take control of the SI bus and transmit the slave address specified above.

c) Wait for an indication of success (slave address sent and target slave responded) by polling or waiting for an interrupt - for details of this stage, refer to Section 12.1.4.

For 10-bit slave address:

a) Call the function **vAHI_SiMasterWriteSlaveAddr()** to indicate that 10-bit slave addressing will be used and to specify the two most significant bits of the relevant slave address. Also, initially specify through this function that a write operation will be performed. This function will put the specified information in the SI master's buffer, but will not transmit it on the SI bus.

b) Call the function **bAHI_SiMasterSetCmdReg()** to issue Start and Write commands, in order to take control of the SI bus and transmit the slave address information specified above.

c) Wait for an indication of success (slave address information sent and at least one matching slave responded) by polling or waiting for an interrupt - for details of this stage, refer to Section 12.1.4.

d) Call the function **vAHI_SiMasterWriteData8()** to specify the eight remaining bits of the slave address. This function will put the specified information in the SI master's buffer, but will not transmit it on the SI bus.

e) Call the function **bAHI_SiMasterSetCmdReg()** to issue a Write command, in order to transmit the slave address information specified above.

f) Wait for an indication of success (slave address information sent and target slave responded) by polling or waiting for an interrupt - for details of this stage, refer to Section 12.1.4.

g) Call the function **vAHI_SiMasterWriteSlaveAddr()** again, indicating that 10-bit slave addressing will be used and specifying the two most significant bits of the relevant slave address. This time, specify through this function that a read operation will be performed on the slave. This function will put the specified information in the SI master's transmit buffer, but will not transmit it on the SI bus.

h) Call the function **bAHI_SiMasterSetCmdReg()** to issue Start and Write commands, in order to take control of the SI bus and transmit the slave address information specified above.

i) Wait for an indication of success by polling or waiting for an interrupt - for details of this stage, refer to Section 12.1.4.

Step 2 **Read data byte from slave**

If only one data byte or the final data byte is to be read from the slave then go directly to Step 3, otherwise follow the instructions below:

a) Call the function **bAHI_SiMasterSetCmdReg()** to issue a Read command, in order to request a data byte from the slave. Also use this function to enable an ACK (acknowledgement) to be sent to the slave once the byte has been received.

b) Wait for an indication of success (read request sent and data received) by polling or waiting for an interrupt - for details of this stage, refer to Section 12.1.4.

c) Call the function **u8AHI_SiMasterReadData8()** to read the received data byte from the SI master's buffer.

Repeat the above instructions (Step 2a-c) for all subsequent data bytes except the final byte to be read (which is covered in Step 3).

Step 3 **Read final data byte from slave**

Read the final (or only) data byte from the slave as follows:

a) Call the function **bAHI_SiMasterSetCmdReg()** to issue Read and Stop commands, in order to request a data byte from the slave and release control of the SI bus. Also use this function to enable a NACK to be sent to the slave once the byte has been received (to indicate that no more data is required).

b) Wait for an indication of success (read request sent and data received) by polling or waiting for an interrupt - for details of this stage, refer to Section 12.1.4.

c) Call the function **u8AHI_SiMasterReadData8()** to read the received data byte from the SI master's buffer.

12.1.4 Waiting for Completion

At various points in the write and read procedures of Section 12.1.2 and Section 12.1.3, it is necessary to wait for an indication of the success of an operation before continuing. The application can use either interrupts or polling to determine when to continue:

- **Interrupts:** SI interrupts can be enabled when **vAHI_SiConfigure()** or **vAHI_SiMasterConfigure()** is called, as described in Section 12.1.1. An SI interrupt (of the type E_AHI_DEVICE_SI) can be generated on a variety of conditions of the Serial Interface. The interrupt is handled by a user-defined callback function registered using the function **vAHI_SiRegisterCallback()**. This interrupt handler should identify the exact source of the SI interrupt and act on it. For more details on the callback function and interrupt sources, refer to Appendix A.1 and Appendix B.2, respectively. In the above write and read procedures, the SI master interrupt source of interest is the one which indicates the completion of a byte transfer or loss of arbitration.
- **Polling:** To determine when the transfer of a byte has finished, the application can regularly call **bAHI_SiMasterPollTransferInProgress()**, which indicates whether a transfer is in progress on the SI bus.

Once an interrupt or polling has indicated that the transfer of a byte has completed, further checks must be made to determine whether the master should stop the data transfer and release the SI bus:

1. In the case of a write to the slave, the application should call the function **bAHI_SiMasterCheckRxNack()** which indicates whether an ACK or a NACK has been received from the slave following the byte transfer:
   - An ACK indicates that the slave can accept more data and therefore further byte transfers can be initiated.
   - A NACK indicates that the slave cannot accept any more data, and that the data transfer must be stopped and the SI bus released.
2. Provided that the SI bus has not already been released, the application should call the function **bAHI_SiMasterPollArbitrationLost()** to check whether the SI master has lost the arbitration of the SI bus. If this is the case, the data transfer must be stopped and the SI bus released.

The data transfer is stopped and the SI bus released by calling the function **bAHI_SiMasterSetCmdReg()** in order to issue the Stop command.

12.2 SI Slave (JN5148 Only)

The SI peripheral on the JN5148 device can act as an SI master or an SI slave (but not as both at the same time). This section describes what must be done to allow the SI slave to participate in a data transfer initiated by a remote SI master.

12.2.1 Enabling the SI Slave and its Interrupts

The SI slave must first be configured and enabled using the function **vAHI_SiSlaveConfigure()**. This function requires the address size of the SI slave to be specified as 7-bit or 10-bit, and the SI slave address itself to be specified. The function also allows the generation of SI slave interrupts to be configured - interrupts can be triggered on the following conditions:

- Data buffer requires data byte for transmission to SI master
- Byte in data buffer sent to SI master and so buffer free for next byte
- Data buffer contains data byte from SI master, available to be read by SI slave
- Final data byte received from SI master (end of data transfer)
- I^2C protocol error

SI interrupts (of the type E_AHI_DEVICE_SI) are handled by the user-defined callback function registered using the function **vAHI_SiRegisterCallback()**. This is the same callback function and registration function as used for an SI master. For details of the callback function prototype, refer to Appendix A.1.

Caution: *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.*

For JN5148, **vAHI_SiSlaveConfigure()** also allows a pulse suppression filter to be enabled, which suppresses any spurious pulses (high or low) with a pulse width less than 62.5 ns on the clock and data lines. Also note that a JN5148 SI slave enabled using this function can later be disabled using **vAHI_SiSlaveDisable()**.

12.2.2 Receiving Data from the SI Master

An SI master indicates that it needs to send data to a particular SI slave as described in Section 12.1.2. The SI slave automatically responds to the SI master according to the protocol for this request, but the application associated with the slave must deal with the data that arrives from the master.

The data transfer on the SI bus consists of a sequence of data bytes, where each byte must be received and then read from the SI slave before the next byte can be received. Interrupts are used to signal the arrival of a data byte from the SI master:

- An interrupt can be generated when a data byte has arrived from the SI master and is available to be read from the SI slave's buffer.
- An interrupt can also be generated when the final data byte of the transfer has arrived from the SI master and is available to be read from the SI slave's buffer.

To use these interrupts, they must have been enabled when the function **vAHI_SiSlaveConfigure()** was called. The registered SI interrupt handler must also deal with them - see Section 12.2.1.

Once a received data byte is available in the SI slave's buffer, it can be read from the buffer by the application using the function **u8AHI_SiSlaveReadData8()**.

12.2.3 Sending Data to the SI Master

An SI master indicates that it needs to obtain data from a particular SI slave as described in Section 12.1.3. The SI slave automatically responds to the SI master according to the protocol for this request, but the application associated with the slave must supply the data that is to be sent to the master.

The data transfer on the SI bus consists of a sequence of data bytes, where each byte must be written to the SI slave's buffer and transmitted before the next byte can be written to the buffer. Interrupts are used to signal when the next data byte is needed in the buffer. To use these interrupts, they must have been enabled when the function **vAHI_SiSlaveConfigure()** was called. The registered SI interrupt handler must deal with them - see Section 12.2.1.

Once a new data byte is required in the SI slave's buffer, it can be written to the buffer by the application using the function **vAHI_SiSlaveWriteData8()**.

13. Serial Peripheral Interface (SPI Master)

This chapter describes control of the Serial Peripheral Interface (SPI) using functions of the Integrated Peripherals API.

The Serial Peripheral Interface on the JN51xx microcontrollers allows high-speed synchronous data transfers between the microcontroller and peripheral devices, without software intervention.

The microcontroller operates as the master on the SPI bus and all other devices connected to the bus are then expected to be slave devices under the control of the master's CPU. The SPI device supports up to five slave devices, one of which is Flash memory (by default).

Dedicated pins are provided for Data In (SPIMISO), Data Out (SPIMOSI) and Clock (SPICLK), which are shared on the SPI bus. A dedicated pin is also provided for Slave-select 0 (SPISEL0), which is assumed to be connected to Flash memory and is read during the boot sequence. Up to 4 more slave-select lines (SPISEL1-SPISEL4) can be used which, if enabled, appear on DIO0 to DIO3.

Data transfer is full-duplex, so data is transmitted by both communicating devices at the same time. Data to be transmitted is stored in a FIFO buffer in the device. The available data transaction sizes depend on the device type:

- **JN5148:** Any transaction size between 1 and 32 bits (inclusive) can be used.
- **JN5139:** A transaction size of 8, 16 or 32 bits can be used.

The data transfer order can be configured as LSB (least significant bit) first or MSB (most significant bit) first.

Since the data transfer is synchronous, both transmitting and receiving devices use the same clock, provided by the SPI master. The SPI device uses the 16-MHz clock, which may be divided down to allow bit rates from 250 kbps to 16 Mbps.

An interrupt can be enabled, which is generated when the data transfer completes.

13.1 SPI Modes

The clock edge on which data is latched is determined by the SPI mode of operation used (0, 1, 2 or 3), which is determined by two boolean parameters, clock polarity and phase, as indicated in the table below.

SPI Mode	Polarity	Phase	Description
0	0	0	Data latched on rising edge of clock
1	0	1	Data latched on falling edge of clock
2	1	0	Clock inverted and data latched on falling edge of clock
3	1	1	Clock inverted and data latched on rising edge of clock

Table 5: SPI Modes of Operation

13.2 Slave Selection

Before transferring data, the SPI master must select the slave(s) with which it wishes to communicate. Thus, the relevant slave-select line(s) must be asserted. It is usual for the SPI master to communicate with a single slave at a time, so not to receive data from multiple slaves simultaneously (unless the slave devices can be inhibited from transmitting data). An 'Automatic Slave Selection' feature is provided, which only asserts the chosen slave-select line(s) during a data transfer.

Manual slave selection is preferred over 'Automatic Slave Selection' when a number of consecutive data transfers are to be performed with a particular slave device, avoiding the need for the slave to be deselected and then reselected between adjacent transfers.

13.3 Using the Serial Peripheral Interface

This section describes how to use the Integrated Peripherals API functions to operate the Serial Peripheral Interface.

13.3.1 Performing a Data Transfer

A SPI data transfer is performed as follows:

1. The SPI master must first be configured using the function **vAHI_SpiConfigure()**. This function allows the configuration of:
   - Number of extra SPI slaves (in addition to Flash memory)
   - Clock divisor (for 16-MHz clock)
   - Data transfer order (LSB first or MSB first)
   - Clock polarity (unchanged or inverted)
   - Phase (latch data on leading edge or trailing edge of clock)
   - Automatic Slave Selection
   - SPI interrupts

   If SPI interrupts are enabled, a corresponding callback function must be registered using the function **vAHI_SpiRegisterCallback()** - see Section 13.4.
2. The SPI slaves must be selected using the function **vAHI_SpiSelect()**. If 'Automatic Slave Selection' is off, the relevant slave-select line(s) will be asserted immediately, otherwise the line(s) will only be asserted during a subsequent data transfer.
3. A data transfer is implemented using **vAHI_SpiStartTransfer()** on JN5148 (for a transaction size between 1 and 32 bits) or using one of the following functions on JN5139 (depending on the transaction size):
   - **vAHI_SpiStartTransfer8()** for 8-bit data
   - **vAHI_SpiStartTransfer16()** for 16-bit data
   - **vAHI_SpiStartTransfer32()** for 32-bit data

4. The transfer is allowed to complete by waiting for a SPI interrupt (if enabled) to indicate completion, or by calling **vAHI_SpiWaitBusy()** which returns when the transfer has completed, or by periodically calling **bAHI_SpiPollBusy()** to check whether the SPI master is still busy.
5. Data received from a slave is read using **u32AHI_SpiReadTransfer32()** on JN5148 or using one of the following functions on JN5139 (depending on the transaction size):
   - **u8AHI_SpiReadTransfer8()** for 8-bit data
   - **u16AHI_SpiReadTransfer16()** for 16-bit data
   - **u32AHI_SpiReadTransfer32()** for 32-bit data
6. If another transfer is required then Steps 3 to 5 must be repeated for the next data. Otherwise, if 'Automatic Slave Selection' is off, the SPI slaves must be de-selected by calling **vAHI_SpiSelect(0)** or **vAHI_SpiStop()**.

A number of other SPI functions exist in the Integrated Peripherals API. The current SPI configuration can be obtained and saved using **vAHI_SpiReadConfiguration()**. If necessary, this saved configuration can later be restored in the SPI using the function **vAHI_SpiRestoreConfiguration()**.

13.3.2 Performing a Continuous Transfer (JN5148 Only)

On the JN5148 device, continuous SPI transfers can be initiated by calling the function **vAHI_SpiContinuous()** instead of **vAHI_SpiStartTransfer()**. This mode facilitates back-to-back reads of the received data, with the incoming data transfers 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 incoming data transfer.

In this case, Steps 1-2 of the procedure in Section 13.3.1 remain the same but Steps 3 and onwards are replaced by the following:

3. A continuous data transfer is started using **vAHI_SpiContinuous()**, which requires the data length (1 to 32 bits) of an individual transfer to be specified.
4. **bAHI_SpiPollBusy()** must be called periodically to check whether the SPI master is still busy with an individual transfer.
5. Once the latest transfer has completed (the SPI master is no longer busy), the the received data from this transfer must be read by calling the function **u32AHI_SpiReadTransfer32()** - the read data is aligned to the right (lower bits) of the 32-bit return value.
6. Once the data has been read, the next transfer will automatically occur and the transferred data must be read as detailed in Steps 4-5 above. However, a continuous transfer can be stopped at any time by calling the function **vAHI_SpiContinuous()** again, this time to disable continuous mode (after this function call, there will be one more transfer before the transfers are stopped).
7. If 'Automatic Slave Selection' is off, after stopping a continuous transfer the SPI slaves must be de-selected by calling **vAHI_SpiSelect(0)**.

13.4 SPI Interrupts

A SPI interrupt can be used to indicate when a data transfer initiated by the SPI master has completed. This interrupt is enabled in **vAHI_SpiConfigure()**.

SPI interrupts are handled by a user-defined callback function, which must be registered using **vAHI_SpiRegisterCallback()**. The relevant callback function is automatically invoked when an interrupt of the type E_AHI_DEVICE_SPIM occurs. For details of the callback function prototype, refer to Appendix A.1.

Caution: *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.*

14. Intelligent Peripheral Interface (SPI Slave)

This chapter describes control of the Intelligent Peripheral (IP) interface using functions of the Integrated Peripherals API.

14.1 IP Interface Operation

The Intelligent Peripheral (IP) interface is used for high-speed data exchanges between the JN51xx microcontroller and a 'remote' processor, which may be a separate processor contained in the wireless network node. The data exchange requires minimal use of the CPU of this processor.

This interface is based on the Serial Peripheral Interface (SPI) - see Chapter 13. The IP interface on the JN51xx microcontroller is a SPI slave - the remote processor must contain the SPI master (which initiates data transfers).

Data transfer is full-duplex, so data is transmitted by both communicating devices at the same time. The JN51xx device uses a Transmit buffer and Receive buffer in a dedicated block of local memory for the data exchanges - each buffer in this IP memory block contains sixty-three 32-bit words. As the master device, the remote processor must initiate the transfer. Data is transmitted and received simultaneously. Only SPI mode 0 is supported, in which data is transmitted on a negative clock edge and received on a positive clock edge.

Tip: Although the data transfer is full-duplex, a simplex transfer can be achieved by transferring dummy data in the unwanted direction.

The interface shares pins with DIO14-18.

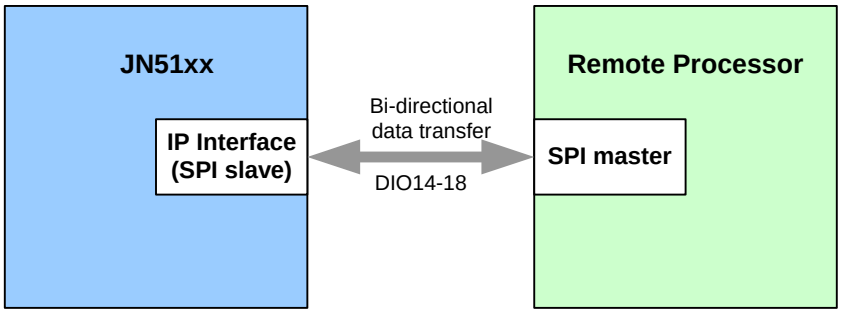


Figure 10: IP Interface as SPI Slave

An interrupt can be enabled, which is generated when the data transfer completes - see Section 14.3.

14.2 Using the IP Interface

A data transfer is conducted via the IP interface (SPI slave) as follows:

1. The IP interface must first be enabled using the function **vAHI_IpEnable()**.
   - Although this function allows the transmit and receive clock edges to be selected, the IP interface only supports SPI mode 0 which requires that data is transmitted on a negative edge and received on a positive edge.
   - This function allows IP interrupts to be enabled that are generated on the completion of data transfers. If enabled, IP interrupts are handled by a callback function registered using **vAHI_IpRegisterCallback()** - see Section 14.3.
2. Once the application is prepared to transmit and/or receive data, one of two functions can be called:
   - **bAHI_IpSendData()** can be called to copy data from RAM into the IP Transmit buffer and to indicate to the remote processor that the JN5148/ JN5139 device is ready to exchange data - that is, either send and receive data at the same time or just send data (in the latter case, the data received in the subsequent bi-directional transfer should be ignored).
   - **vAHI_IpReadyToReceive()** can be called on the JN5148 device (only) to indicate to the remote processor that the local device is ready to receive data (the data sent in the subsequent bi-directional transfer should then be ignored by the remote processor).

   It is then the responsibility of the remote processor, as the SPI master, to initiate the data transfer.
3. The data transfer is allowed to complete by waiting for an IP interrupt (if enabled) to indicate completion. Alternatively, two functions can be periodically called to check whether the data transfer has completed:
   - **bAHI_IpTxDone()** can be used to check whether all data has been transmitted.
   - **bAHI_IpRxDataAvailable()** can be used to check whether data has been received.
4. Once the received data is available, it can be copied from the IP Receive buffer into RAM using the function **bAHI_IpReadData()**. Subsequent behaviour depends on the local device type:
   - On JN5139, the above function automatically indicates to the remote processor that a new transfer can be initiated. The application should then return to Step 3 to wait for the next transfer to complete.
   - On JN5148, the application should return to Step 2 when it is ready for the next transfer (this allows time between transfers, e.g. for data processing).

Note: The byte order of data to be sent must be specified as Big Endian or Little Endian. This is done in the function **vAHI_IpEnable()** for JN5139 and in **bAHI_IpSendData()** for JN5148.

14.3 IP Interrupts

An IP interrupt can be used to indicate when a data transfer has completed. This interrupt is enabled in **vAHI_IpEnable()**.

IP interrupts are handled by a user-defined callback function, which must be registered using **vAHI_IpRegisterCallback()**. The relevant callback function is automatically invoked when an interrupt of the type E_AHI_DEVICE_INTPER occurs. For details of the callback function prototype, refer to Appendix A.1.

Caution: *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.*

15. Digital Audio Interface (DAI) [JN5148 Only]

This chapter describes control of the Digital Audio Interface (DAI) of the JN5148 device using functions of the Integrated Peripherals API.

The JN5148 device contains a 4-wire Digital Audio Interface which allows communication with external devices that support other digital audio interfaces, such as CODECs.

Note: The data path between the CPU and the DAI can be optionally buffered using the Sample FIFO interface, described in Chapter 16. Also refer to Section 15.3.

15.1 DAI Operation

The Digital Audio Interface on the JN5148 device is compatible with the industry-standard I²S interface and acts as the interface master. The signals, data format and data transfer modes supported by the interface are described in the sub-sections below.

15.1.1 DAI Signals and DIOs

The DAI is a 4-wire interface that uses four of the DIO pins of the JN5148 device. The DAI signals and corresponding DIO pins are detailed in the table below.

DAI Signal	DIO Pin	Signal Description
SDIN	13	**Data In:** Audio data is received on this line.
SDOUT	18	**Data Out:** Audio data is transmitted on this line.
WS	12	**Word Select:** Indicates for which stereo channel (Left or Right) data is currently being transferred - normally: • asserted (1) for Right channel • de-asserted (0) for Left channel It is possible to invert WS, so that 0 is for Right and 1 is for Left.
SCK	17	**Clock:** Bit clock for transfer of audio data. This is derived from the 16-MHz system clock and the clock frequency is configurable.

Table 6: DAI Signals and DIO Pins

Note that the data transfer is always full-duplex, so audio data will be transmitted on SDOUT and received on SDIN simultaneously.

15.1.2 Audio Data Format

Audio data is normally serially transferred (on the SDOUT and SDIN lines) with up to 16 bits per stereo channel. It is possible to have fewer than 16 bits of actual audio data per channel and to (optionally) make up the number of bits per channel to 16 by padding with zeros.

It is also possible to implement data transfers with more than 16 bits per channel - up to 32 bits per channel, in fact. In this case, the actual audio data can still only occupy a maximum of 16 bits per channel and zero-padding must be enabled for those bits beyond the basic 16 bits.

The audio data format described above is summarised in Figure 11 below.

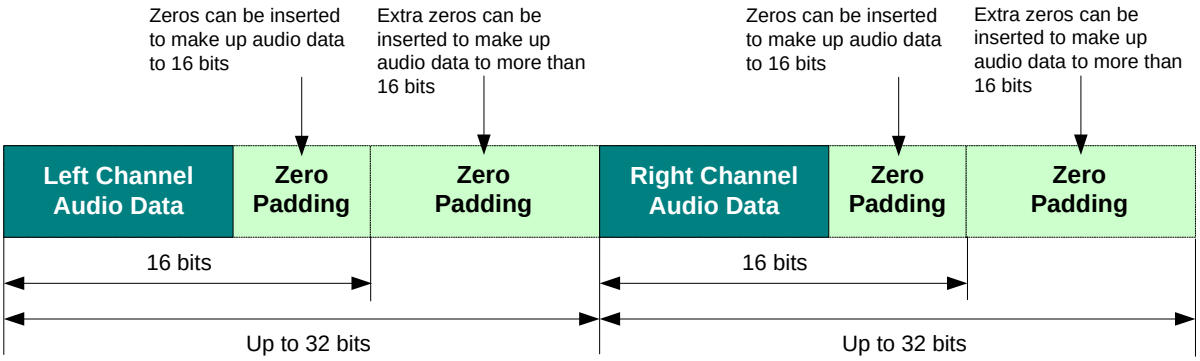


Figure 11: Format of Transferred Audio Data

15.1.3 Data Transfer Modes

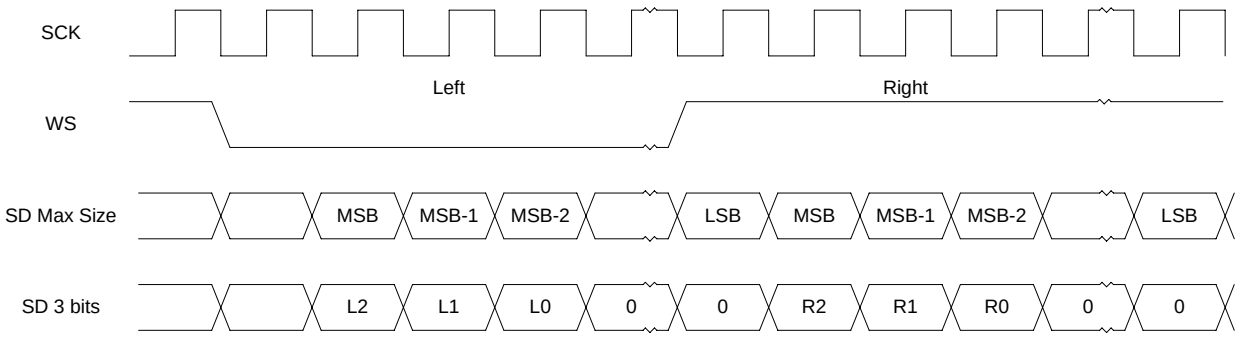
An audio data frame is always transferred from the DAI with left channel first and right channel second. Within a channel, the audio data is transferred starting with the most significant bit (MSB), although this bit may not be the first bit actually transferred (see modes below). The DAI will always transfer both left- and right-channel data. In the transfer of mono data, one channel is unused and should be padded out with zeros during transmission - similarly, the bits for the unused channel should be ignored during reception. There are three possible DAI modes, each based on this format.

I²S Mode

The format of the audio data transfer in I²S mode is as follows:

- During idle periods, the WS line takes its state for the right channel - that is, the ‘asserted’ state. During a frame transfer, the WS line is then de-asserted just before the left-channel data and is re-asserted just before the right-channel data.
- The MSB of the left-channel data is transferred one clock cycle (SCK line) after the WS transition and the MSB of the right-channel data is transferred one clock cycle after the next (opposite) WS transition. Within a channel, any zero-padding is added after the actual audio data.

An audio data frame transfer in I²S mode is illustrated in Figure 12 below.



This example assumes that the channel data comprises 3 bits: L2 L1 L0 for left channel, R2 R1 R0 for right channel.

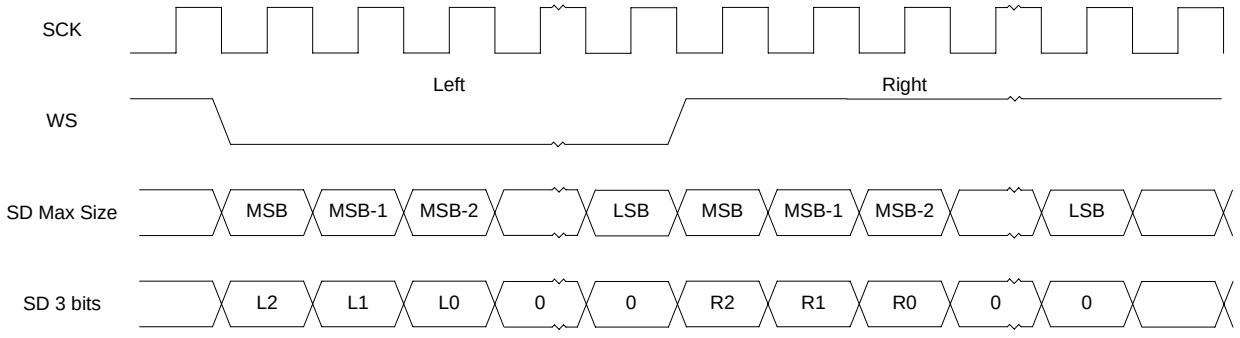
Figure 12: I²S Transfer Mode

Left-justified Mode

The DAI can operate in left-justified mode, if required. In this mode:

- The polarity of the WS signal can be optionally inverted.
- During idle periods, the WS line normally takes its state for the right channel (as in I²S mode). During a frame transfer, there is then a transition of the WS signal at the start of the left-channel data and then an opposite transition at the start of the right-channel data.
- The data bits are aligned such that the MSB of the left-channel data is transferred on the same clock cycle (SCK line) as the WS transition and the MSB of the right-channel data is transferred on the same clock cycle as the next (opposite) WS transition. Within a channel, any zero-padding is added after the actual audio data.

An audio data frame transfer in left-justified mode is illustrated in Figure 13 below (in this example, the WS signal has not been inverted).



This example assumes that the channel data comprises 3 bits: L2 L1 L0 for left channel, R2 R1 R0 for right channel.

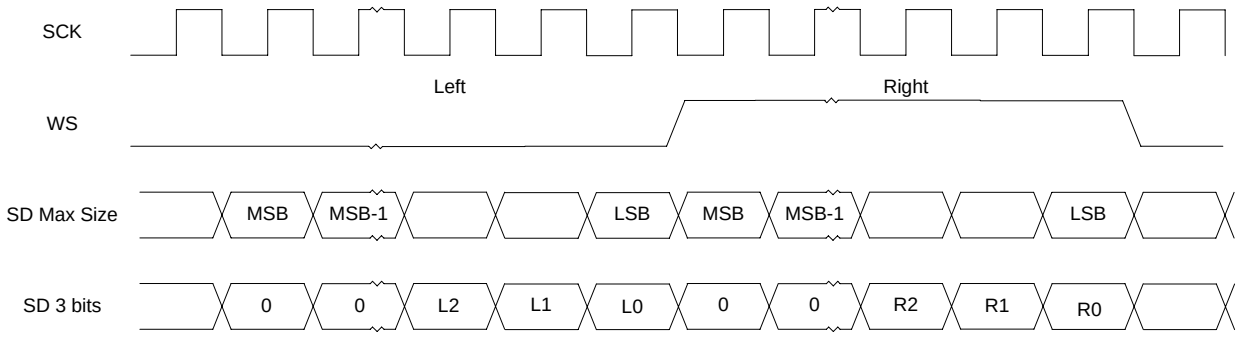
Figure 13: Left-justified Mode

Right-justified Mode

The DAI can operate in right-justified mode, if required. In this mode:

- The polarity of the WS signal can be optionally inverted.
- During idle periods, the WS line normally takes its state for the left channel. During a frame transfer, there is then a transition of the WS signal at the start of the right-channel data and then an opposite transition at the end of the right-channel data.
- The data bits are aligned such that the MSB of the right-channel data is transferred on the same clock cycle (SCK line) as the WS transition and the LSB of the right-channel data is transferred on the clock cycle before the next (opposite) WS transition. Within a channel, any zero-padding is added before the actual audio data.

An audio data frame transfer in right-justified mode is illustrated in Figure 14 below (in this example, the WS signal has not been inverted).



This example assumes that the channel data comprises 3 bits: L2 L1 L0 for left channel, R2 R1 R0 for right channel.

Figure 14: Right-justified Mode

15.2 Using the DAI

This section describes how to use the Integrated Peripherals API functions to operate the Digital Audio Interface.

15.2.1 Enabling the DAI

The DAI must first be powered on using the function **vAHI_DaiEnable()**. This function can also be used to power down the DAI, when required.

15.2.2 Configuring the Bit Clock

The DAI bit clock is derived from the 16-MHz system clock and is transmitted on the SCK line to provide bit synchronisation when transferring audio data. The bit clock must be configured using the function **vAHI_DaiSetBitClock()** in the following ways:

- The system clock is scaled to produce a bit clock frequency in the range 8 MHz to approximately 127 kHz. To achieve this, the 16-MHz source frequency is divided by an even integer value in the range 2 to 126, where this scaling is specified using the above function. The default bit clock frequency is 1 MHz.
- The clock output on the SCK line can be enabled permanently or only during data transfers. This choice is made using the above function.

15.2.3 Configuring the Data Format

Data transfers via the DAI must be configured in terms of data size/padding and the transfer mode. These configurations are described separately below. The required settings depend on the external device to which the DAI is connected.

Data Size/Padding

The number of audio data bits per channel can be up to 16, although the total number of bits per channel can be up to 32. Any bits that are not used for audio data must be set to zero. This is described in Section 15.1.2.

The function **vAHI_DaiSetAudioData()** is used to configure data size and zero-padding per channel, as follows:

- The number of audio data bits per channel must be specified in the range 1 to 16.
- If there are fewer than 16 audio data bits per channel, an option can be enabled to automatically make up the total number of bits per channel to 16 by adding zeros.
- If the required total number of bits per channel is greater than 16, an option can be enabled to automatically add the relevant number of extra zero-padding bits (in addition to those required to pad to 16 bits). Up to 16 extra zero-padding bits can be added (to achieve a maximum of 32 bits per channel).

For example, if there are 12 bits of audio data per channel but a total of 24 bits per channel are required, 4 zero-bits are added to make the number of bits up to 16 and 8 extra zero-bits are added to make the total up to 24.

Transfer Mode

A data transfer can operate in one of three modes (I²S, left-justified or right-justified), described in Section 15.1.3. Within each mode, choices are available. The mode is selected and configured using the function **vAHI_DaiSetAudioFormat()**, as follows:

- The operating mode can be selected as I²S, left-justified or right-justified.
- The polarity of the WS signal can be inverted (must not be done for I²S mode).
- The WS state during idle time can be configured to be its left-channel state or its right-channel state (must be set as right-channel state for I²S mode).

15.2.4 Enabling DAI Interrupts

An interrupt can be generated on completion of each data transfer from/to the DAI. If DAI interrupts are to be used, they must be enabled using the function **vAHI_DaiInterruptEnable()**. In addition, a user-defined callback function to handle the interrupts (of the type E_AHI_DEVICE_I2S) must be registered using the function **vAHI_DaiRegisterCallback()**. For details of the callback function prototype, refer to Appendix A.1.

Caution: *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.*

The use of DAI interrupts is described further in Section 15.2.5 below.

15.2.5 Transferring Data

As the interface master, the DAI on the JN5148 device initiates data transfers (under the control of the application). These transfers are full-duplex, so the DAI transmits and receives data at the same time. A single data frame (containing left-channel and right-channel audio data) is transmitted and received during an individual transfer.

Note: This section describes data transfers using the DAI Transmit and Receive buffers. Alternatively, the DAI can be connected to the Sample FIFO interface of the JN5148 device. In this case, the function calls described in this section are not applicable. Use of the DAI with the Sample FIFO interface is outlined in Section 15.3.

Preparing a Data Transfer

The data frame to be transmitted must be stored in the DAI Transmit buffer. When received, an incoming data frame will be stored in the DAI Receive buffer. Before starting a data transfer, the application must prepare these buffers:

- The data frame to be transmitted must be loaded into the Transmit buffer using the function **vAHI_DaiWriteAudioData()**. The left-channel data and right-channel data are passed into this function via separate 16-bit parameters.
- The Receive buffer should be free of the previous data frame that was received. The application can ensure that the Receive buffer is clear by calling the function **vAHI_DaiReadAudioData()** to read any remaining data in the buffer.

Performing the Data Transfer

A data transfer can be started by calling the function **vAHI_DaiStartTransaction()**.

Once the transfer has completed, the received data frame can be obtained from the DAI Receive buffer by calling **vAHI_DaiReadAudioData()** - the left-channel data and right-channel data are obtained from this function via separate returned pointers. The application can also prepare the next transfer by calling **vAHI_DaiWriteAudioData()** to load the next data frame to be transmitted into the DAI Transmit buffer. The next data transfer can then be initiated by calling **vAHI_DaiStartTransaction()**.

The timing of the above set of read/write/start function calls can be controlled in one of three ways, described below:

- **Polling:**

  The function **bAHI_DaiPollBusy()** can be called on a regular basis to check whether the DAI is still performing the previous transfer. Once this function returns FALSE, the next read/write/start function calls can be made.

- **DAI Interrupts:**

  A DAI interrupt can be used to signal when the previous transfer has completed. This interrupt must have been enabled using **vAHI_DaiInterruptEnable()**. The generated interrupt is of the type E_AHI_DEVICE_I2S, which will be automatically handled by the registered callback function for DAI interrupts - see Section 15.2.4. Once this interrupt has occurred, the next read/write/start function calls can be made.

- **Timer Interrupts:**

  A JN5148 timer (Timer 0, 1 or 2) can be used to schedule data transfers at regular intervals. Interrupts must be enabled for the timer and on each timer interrupt, the next read/write/start function calls can be made. Timers and timer interrupts are described in Chapter 7. Care must be taken to allow enough time for an individual transfer to complete before the next timer interrupt is generated.

15.3 Using the DAI with the Sample FIFO Interface

Normally, the DAI Transmit and Receive buffers are used to store audio data between the CPU and DAI, where each buffer holds a single data frame containing left-channel and right-channel audio data. Alternatively, the Sample FIFO interface (described in Chapter 15) can be used to hold audio data between the CPU and DAI.

The Sample FIFO interface comprises transmit and receive paths, each containing a FIFO able to store ten 16-bit words. This interface is only able to handle 16-bit mono audio data, where up to 10 mono audio samples can be stored in each FIFO. The advantage of using this interface is that each CPU read/write operation can comprise up to 10 mono audio samples (each way) rather than a single stereo audio sample, thereby requiring less regular CPU intervention. The scheduling of the transfers between the FIFOs and DAI is provided by Timer 2 operating in 'Timer repeat' mode (see Chapter 7), such that a transfer is initiated every time the timer produces a rising signal.

Although the Sample FIFO interface can only store 16-bit mono audio samples, each mono sample will be transferred between the DAI and external device in a stereo data frame. The 16-bit mono sample can be transported in either the left channel or the right channel of the data frame.

To use the DAI in conjunction with the Sample FIFO interface (with Timer 2), refer to Chapter 16 - an example procedure is given in Section 16.3.

16. Sample FIFO Interface (JN5148 Only)

This chapter describes control of the Sample FIFO interface of the JN5148 device using functions of the Integrated Peripherals API.

The Sample FIFO interface comprises transmit and receive paths, each containing a FIFO able to store ten 16-bit words. This interface is primarily designed to buffer audio data between the CPU and the Digital Audio Interface (DAI), described in Chapter 15 (although these FIFOs are not essential to the operation of the DAI). Therefore, particular reference is made to the DAI in the description of the Sample FIFO interface in this chapter. Use of the DAI in conjunction with the Sample FIFO interface is also described in Section 15.3.

16.1 Sample FIFO Operation

The Sample FIFO interface allows up to ten 16-bit words to be buffered on their way to or from the CPU of the JN5148 device. This interface can reduce the frequency at which the CPU needs to generate output data and/or process input data, which may allow more efficient CPU operation.

The Sample FIFO interface is illustrated in Figure 15 below.

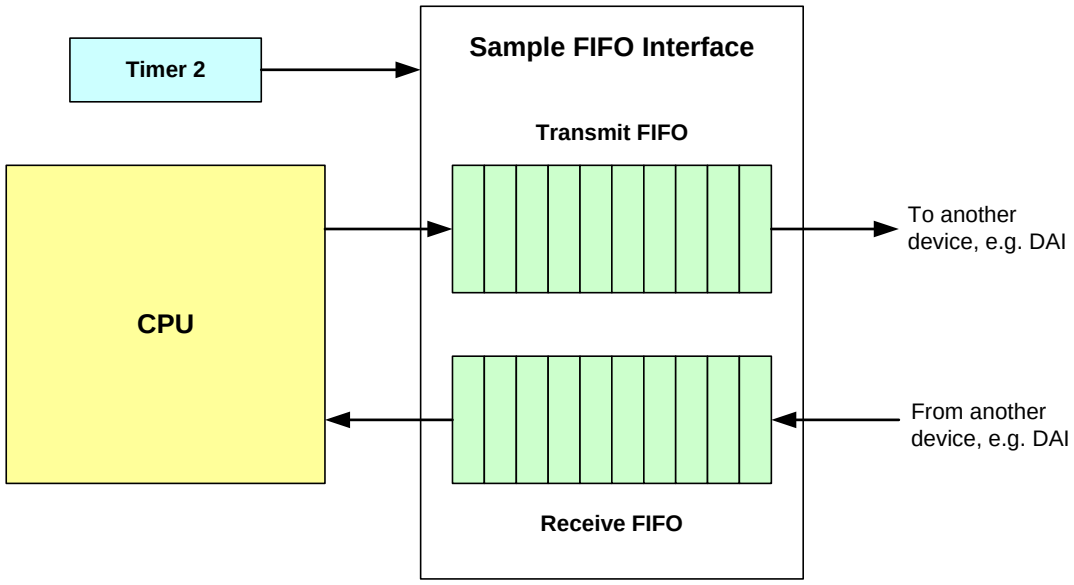


Figure 15: Sample FIFO Interface

On the DAI side of the FIFOs, the input and output of data are governed by Timer 2, which must be set to run in 'Timer repeat' mode (see Section 7.3.1). Data input/output automatically occurs on every rising edge of the pulsed signal generated by Timer 2. On this Timer 2 trigger, one sample of audio data is passed (in each direction) between the FIFOs and the DAI, and the DAI also exchanges data with the external device to which it is connected.

Note: The Sample FIFOs are only able to store 16-bit mono audio data, although the DAI external transfer will be made in terms of stereo audio data frames containing left and right channels. In practice, a mono sample is stored in one stereo channel of a transferred frame.

On the CPU side of the FIFOs, the CPU (application) must write a burst of data to the Transmit FIFO and read a burst of data from the Receive FIFO at appropriate times. Interrupts can be used to aid the timings of these CPU read and write operations. Sample FIFO interrupts can be generated when:

- the Transmit FIFO fill-level falls below a pre-defined threshold - can be used to prompt a write to the FIFO to provide further data to be transmitted
- the Transmit FIFO becomes empty - can be used to prompt a write to the FIFO to provide further data to be transmitted
- the Receive FIFO fill-level rises above a pre-defined threshold - can be used to prompt a read of the FIFO to collect received data
- the Receive FIFO becomes full and an attempt to add more data fails (overflow) - this is an error condition, resulting in lost data, and prompts a read of the FIFO to make space for new data

16.2 Using the Sample FIFO Interface

This section describes how to use the Integrated Peripherals API functions to operate the Sample FIFO interface.

16.2.1 Enabling the Interface

The Sample FIFO interface must first be enabled using the function **vAHI_FifoEnable()**. This function can also be used to disable the interface, when required.

16.2.2 Configuring and Enabling Interrupts

Interrupts can be used to prompt the application to write/read data to/from the Sample FIFO interface (see Section 16.1). These interrupts can be enabled using the function **vAHI_FifoEnableInterrupts()**, which allows four different interrupts (already outlined in Section 16.1) to be individually enabled/disabled:

- **Transmit interrupt:** Generated when the number of samples in the Transmit FIFO falls below a level which is pre-defined using the function **vAHI_FifoSetInterruptLevel()**.
- **Transmit Empty interrupt:** Generated when the Transmit FIFO becomes empty.
- **Receive interrupt:** Generated when the number of samples in the Receive FIFO rises above a level which is pre-defined using the function **vAHI_FifoSetInterruptLevel()**.
- **Receive Overflow interrupt:** Generated when the number of samples in the Receive FIFO reaches the maximum capacity of the FIFO (10 samples) and an attempt has been made to add more samples.

The function **vAHI_FifoSetInterruptLevel()** also allows selection of the device which is to be connected to the Sample FIFO interface (currently, the only option is the DAI).

In addition, a user-defined callback function to handle the interrupts (of the type E_AHI_DEVICE_AUDIOFIFO) must be registered using the function **vAHI_FifoRegisterCallback()**. For details of the callback function prototype, refer to Appendix A.1.

Caution: *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.*

16.2.3 Configuring and Starting the Timer

Timer 2 must be used to schedule the movement of data between the Sample FIFO interface and the connected peripheral device (normally the DAI). This timer must be put into 'Timer repeat' mode to generate a train of pulses - one sample of data will be shipped into and out of the FIFOs on every rising edge of this pulse train.

Note: The data movement scheduled by Timer 2 does not apply to data transfers between the CPU and the Sample FIFO interface. CPU read and write operations on the FIFOs are described in Section 16.2.4.

The timer is configured and started as detailed in Chapter 7, but the following requirements should be noted:

- In the **vAHI_TimerEnable()** function call:
  - the timer output option must be disabled, since the timer will operate in the basic 'Timer' mode (although a PWM signal will be produced by the timer, there will be no need to externally output this signal)
  - interrupts should be disabled for this timer
- In the **vAHI_TimerConfigureOutputs()** function call, external gating must be disabled.
- The timer must be started in 'repeat' mode by calling the function **vAHI_TimerStartRepeat()** (which also allows the period of the pulsed signal to be defined).

16.2.4 Buffering Data

The Sample FIFO interface facilitates the buffering of 16-bit data samples between the CPU and another peripheral device (the DAI), allowing samples to be moved in blocks of up to 10 (in each direction). As described in Section 16.1 and Section 16.2.3, duplex data transfers between the FIFOs and the peripheral device (DAI) are automatically triggered by Timer 2. However, the data transfers between the CPU and the FIFOs must be explicitly controlled by the application, as described below.

The cases of writing to and reading from the Sample FIFO interface are dealt with separately below.

Writing Data to FIFO

Before the application writes data to the Sample FIFO interface, it should call the function **u8AHI_FifoReadTxLevel()** to obtain the number of data samples currently in the Transmit FIFO. Provided the FIFO is not full, the function **vAHI_FifoWrite()** can then be called to write data to the FIFO - this function writes a single 16-bit data sample on each call and must therefore be called multiple times according to the number of samples to be written.

Reading Data from FIFO

Before the application reads data from the Sample FIFO interface, it should call the function **u8AHI_FifoReadRxLevel()** to obtain the number of data samples currently in the Receive FIFO. Provided the FIFO is not empty, the function **bAHI_FifoRead()** can then be called to read data from the FIFO - this function reads a single 16-bit data sample on each call and must therefore be called multiple times according to the number of samples available to be read.

16.3 Example FIFO Operation

This section outlines a typical use of the Sample FIFO interface to pass 16-bit mono audio data samples to and from the DAI. In this example, the FIFOs are serviced by the CPU when the number of samples in the Transmit FIFO falls to 2 and the number of samples in the Receive FIFO rises to 8.

The procedure below describes the actions to be taken by the CPU application.

Step 1 **Enable, configure and connect the DAI and the Sample FIFO interface**

a) Call **vAHI_DaiEnable()** to enable the DAI and then call **vAHI_FifoEnable()** to enable the Sample FIFO interface.

b) Configure the bit clock for the DAI, as described in Section 15.2.2.

c) Configure the data format for the DAI, as described in Section 15.2.3.

d) Call **vAHI_DaiConnectToFIFO()** to connect the DAI to the Sample FIFO interface - this function requires you to specify whether the 16-bit mono data will be contained in the left channel or right channel of the transferred stereo data frame.

Step 2 **Pre-fill the Transmit FIFO**

a) Check whether there are already any samples in the Transmit FIFO by calling **u8AHI_FifoReadTxLevel()**.

b) Use multiple calls to **vAHI_FifoWrite()** to write the appropriate number of samples to the Transmit FIFO in order to make up the total number of samples in the FIFO to 10.

Step 3 **Empty the Receive FIFO**

a) Check whether there are already any samples in the Receive FIFO by calling **u8AHI_FifoReadRxLevel()**.

b) Use multiple calls to **bAHI_FifoRead()** to read the appropriate number of samples from the Receive FIFO in order to empty the FIFO.

Step 4 **Set the Transmit interrupt level and enable FIFO interrupts**

a) Use **vAHI_FifoSetInterruptLevel()** to set the Transmit FIFO interrupt level to 3 samples and the Receive FIFO interrupt level to 7 samples.

b) Call **vAHI_FifoEnableInterrupts()** to enable the Sample FIFO interface interrupts (you should also have registered a corresponding callback function via **vAHI_FifoRegisterCallback()**).

Step 5 **Enable and Start Timer 2**

a) Call **vAHI_TimerEnable()** to enable Timer 2 - choose an appropriate clock divisor, do not enable Timer interrupts and do not enable the PWM output.

b) Call **vAHI_TimerConfigureOutputs()** to disable external gating.

c) Call **vAHI_TimerStartRepeat()** to start the timer in ‘repeat mode’ with the appropriate period for the desired data transmission rate.

Step 6 **Wait for a FIFO interrupt and service the interrupt**

a) Wait for an interrupt of the type E_AHI_DEVICE_AUDIOFIFO to occur (which will invoke the registered callback function).

b) In the callback function, use multiple calls to **vAHI_FifoWrite()** to write 8 new samples to the Transmit FIFO.

c) Also in the callback function, use multiple calls to **bAHI_FifoRead()** to read 8 samples from the Receive FIFO.

d) Return from the callback function to **Step 6a**.

17. External Flash Memory

This chapter describes control of external Flash memory using functions of the Integrated Peripherals API.

A JN51xx microcontroller is normally connected to an external Flash memory device which is used to store the binary application and associated application data. The two devices are typically resident on the same carrier board or module.

The Integrated Peripherals API includes functions that allow the application to erase, programme and read a sector of the attached Flash memory. Normally, these functions are used to store and retrieve application data - this might include data to be preserved in non-volatile memory before going to sleep without RAM held.

17.1 Flash Memory Organisation and Types

JN51xx modules are supplied with Flash memory devices fitted, but the API functions can also be used with custom modules and boards which have different Flash devices.

Flash memory is partitioned into sectors. The number of sectors depends on the Flash device type (see Table 7), but the application binary is normally stored from the start of the first sector, denoted Sector 0, and the application data is stored in the final sector.

A Flash memory sector which is blank (no data) comprises entirely of binary 1s. When data is written to the sector, the relevant bits are changed from 1 to 0.

The following table lists the Flash device types supported by JN51xx microcontrollers and gives the number of sectors for each device as well as the size of a sector.

Flash Device	Number of Sectors	Sector Size (Kbytes)
AT25F512	2	32
SST25V010	4	32
M25P10A	4	32
M25P40	8	64

Table 7: Supported Flash Devices

Thus, the supported Flash memory devices are 64-Kbyte, 128-Kbyte or 512-Kbyte in size. Custom Flash devices can also be used.

17.2 Function Types

Some Flash functions of the Integrated Peripherals API are available in two versions:

- One version is designed to interact with a 4-sector 128-Kbyte Flash device in which the application data is stored in Sector 3, e.g. the ST M25P10 device. These functions are designed to access Sector 3 only and all addresses are offsets from the start of this sector.
- The other version is designed to interact with a 128-Kbyte or 512-Kbyte Flash device. These functions are able to access any sector - you should refer to the datasheet for the Flash device to obtain the necessary sector details.

Caution: *Be careful not to erase essential data such as the 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-Kbyte device, and in Sectors 0 and 1 of a 512-Kbyte device.*

17.3 Operating on Flash Memory

This section describes how to use the Flash functions of the Integrated Peripherals API to erase, read from and write to a sector of Flash memory.

The first Flash function called must be the initialisation function **bAHI_FlashInit()**. This function requires the attached Flash device type to be specified, although an auto-detect option for the device type is also available.

A custom Flash device can also be specified. In this case, a set of custom functions must be provided that will be used by the API to access the Flash device.

17.3.1 Erasing Data from Flash Memory

Erasing a portion of Flash memory involves setting any 0 bits to 1. Two functions are provided that allow an entire sector of Flash memory to be erased:

- **bAHI_FlashErase()** can be used on a JN5139 device to erase the final sector of a 4-sector 128-Kbyte Flash device. Only Sector 3 is erased by this function - no other sectors are affected.
- **bAHI_FlashEraseSector()** can be used on a JN5148 or JN5139 device to erase one sector of the attached Flash device. Any sector can be erased and thus care must be taken not to erase the application code.

17.3.2 Reading Data from Flash Memory

Two functions are provided that allow data to be read from a sector of Flash memory:

- **bAHI_FlashRead()** can be used on a JN5139 device to read from the final sector of a 4-sector 128-Kbyte Flash device. Only Sector 3 can be accessed by this function.
- **bAHI_FullFlashRead()** can be used on a JN5148 or JN5139 device to read data from any sector of the attached Flash device.

In either case, the function can be used to read a portion of data starting at any point within the sector.

17.3.3 Writing Data to Flash Memory

Before writing the first data to a sector of Flash memory, the sector must be blank (consisting entirely of binary 1s), as the write operation will only change 1s to 0s (where relevant). Therefore, it may be necessary to erase the relevant sector, as described in Section 17.3.1, before writing the first data to it.

Two functions are provided that allow data to be written within a sector of Flash memory:

- **bAHI_FlashProgram()** can be used on a JN5139 device to write to the final sector of a 4-sector 128-Kbyte Flash device. Only Sector 3 can be accessed by this function.
- **bAHI_FullFlashProgram()** can be used on a JN5148 or JN5139 device to write data to any sector of the attached Flash device.

In either case, the function can be used to write a portion of data starting at any point within the sector. When adding data to existing data in a sector, you must be sure that the relevant portion of the sector is already blank (comprising all binary 1s).

Tip: One way to ensure that data is added successfully to a sector is: first read the entire sector into RAM (see Section 17.3.2), then erase the entire sector in Flash memory (see Section 17.3.1), then add the new data to the existing data in RAM, and finally write all of this data back to the sector in Flash memory.

17.4 Controlling Power to Flash Memory

Flash memory can be optionally powered off while the JN51xx microcontroller is in Sleep mode, and is always automatically powered off for Deep Sleep mode. An unpowered Flash device during sleep allows greater power savings and extends battery life.

Two functions are provided for controlling power to the Flash memory device, but these are only applicable to the following devices:

- STM25P10A attached to a JN5139 or JN5148 device
- STM25P40 attached to a JN5148 device

Calling these functions for other Flash devices will have no effect.

The necessary function calls before and after sleep are outlined below.

Before Sleep

The above Flash memory devices can be powered down before entering sleep mode by calling the function **vAHI_FlashPowerDown()**. This function must be called before **vAHI_Sleep()** is called.

Note 1: In the case of sleep without RAM held, the function **vAHI_FlashPowerDown()** should not be called until all the application data that needs to be preserved during sleep has been saved to Flash memory.

Note 2: There is no need to call the function **vAHI_FlashPowerDown()** for Deep Sleep mode, as the Flash memory device is automatically powered down before entering this mode.

After Sleep

If a Flash memory device was powered down using **vAHI_FlashPowerDown()** before entering sleep with RAM held, on waking from sleep the function **vAHI_FlashPowerUp()** must be called to power on the Flash memory device again.

In the cases of sleep without RAM held and Deep Sleep mode, there is no need to call **vAHI_FlashPowerUp()** on waking, since the Flash memory device is powered on automatically.

Tip: In order to conserve power, you may wish to power down the Flash memory device at JN5148/JN5139 start-up and only power up the Flash device when required.

Part II:
Reference Information

18. 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 JN51xx microcontroller. Note that many of these functions 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.

Note: For guidance on using these functions in JN5148/ JN5139 application code, refer to Chapter 2.

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

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

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 JN51xx 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 JN51xx high-power module.

***Caution:** A JN51xx high-power module cannot be used in channel 26 of the 2.4-GHz band.*

Note that this function cannot be used with a JN51xx high-power module from a manufacturer other than NXP/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 JN5148 high-power module from a manufacturer other than NXP/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 JN5139 device 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 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 Section 3.1) 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

19. System Controller Functions

This chapter describes the functions that interface to the System Controller on the JN51xx microcontroller. 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

Note: For information on the above chip features and guidance on using the System Controller functions in JN5148/JN5139 application code, refer to Chapter 3.

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

u8AHI_PowerStatus

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

Description

This function returns power domain status information for the JN51xx 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 logic is operational - clock is enabled

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 logic is operational
4-7	Unused

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 device.

The function returns when the CPU re-starts.

Parameters

None

Returns

None

vAHI_Sleep

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 143), 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 373.
- 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 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 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 clock. Upon switching the clock on again, the MAC settings must be restored from the saved settings. You can save and restore the MAC settings using functions of the 802.15.4 Stack API:

- To save the MAC settings, use the function **vAppApiSaveMacSettings()**.
- Switching the 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 clock)

The MAC settings save and restore functions are described in the *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*.

While the Digital Logic domain 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 Digital Logic domain clock: TRUE to switch the clock ON FALSE to switch the 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()**.

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()**.

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 or above	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 or above)

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 or above	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 151.

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 data sheet for your 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 10 on page 382

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

20. Analogue Peripheral Functions

This chapter describes the functions that are used to control the analogue peripherals of the JN51xx 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 20.1
- ADC functions, described in Section 20.2
- DAC functions, described in Section 20.3
- Comparator functions, described in Section 20.4

Note: For information on the analogue peripherals and guidance on using these functions in JN5148/JN5139 application code, refer to Chapter 4.

20.1 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:

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 both the JN5148 and JN5139 devices.

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 Section 3.5).

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 Section 4.4.

Parameters

prApCallback	Pointer to callback function to be registered

Returns

None

20.2 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:

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

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

Description

This function can be used 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

20.3 DAC Functions

This section describes the functions that can be used to control the on-chip DACs (Digital-to-Analogue Converters). The JN51xx microcontrollers feature two DACs, denoted DAC1 and DAC2. On the JN5148 device, 12-bit DACs are used, while on the JN5139 device, 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:

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 device, 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 the JN5139 device, 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 device. 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 device - see Caution 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 device, 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 device, 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

20.4 Comparator Functions

This section describes the functions that can be used to control the on-chip comparators. On both the JN5139 and JN5148 devices, there are two comparators (1 and 2).

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

vAHI_ComparatorEnable

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

Description

This function configures and enables the specified comparator. 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

void vAHI_ComparatorDisable(uint8 *u8Comparator***);**

Description

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

Parameters

u8Comparator	Identity of comparator: E_AHI_AP_COMPARATOR_1 E_AHI_AP_COMPARATOR_2

Returns

None

vAHI_ComparatorLowPowerMode

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

Description

This function can be used to enable or disable low-power mode on the comparators of the JN5148/JN5139 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

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

Description

This function enables interrupts for the specified comparator on the JN5148/JN5139 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

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

Description

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

```
uint8 u8AHI_ComparatorWakeStatus(void);
```

Description

This function returns the wake-up interrupt status of the comparators on the JN5148/ JN5139 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.

Parameters

None

Returns

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

21. DIO Functions

This chapter describes the functions that can be used to control the digital input/output lines, referred to as DIOs. The JN51xx 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 microcontroller.

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

Note: For guidance on using the DIO functions in JN5148/JN5139 application code, refer to Chapter 5.

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

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 enabled
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.

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.

22. UART Functions

This chapter details the functions for controlling the on-chip UARTs (Universal Asynchronous Receiver Transmitters). The JN51xx 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.

Note: For information on the UARTs and guidance on using the UART functions in JN5148/JN5139 application code, refer to Chapter 6.

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

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. 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, 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

```
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. 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

23. Timer Functions

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 has 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.

Note: For information on the timers and guidance on using the timer functions in JN5148/JN5139 application code, refer to Chapter 7.

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

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 2 on page 70.

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 2 on page 70):

- 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 2 on page 70):

- 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 2 on page 70.

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 2 on page 70.

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

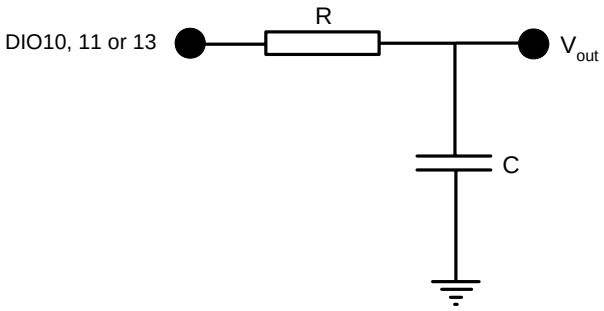
vAHI_TimerStartDeltaSigma

```
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 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

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

Description

This function enables/disables DIOs for use by the specified timer (Timer 0 or 1):

- 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

24. Wake Timer Functions

This chapter details the functions for controlling the wake timers. The JN51xx 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 device.

The wake timers are normally used 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 150 (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.

Note: For guidance on using the wake timer functions in JN5148/JN5139 application code, refer to Chapter 8.

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

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 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 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 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 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 0x7FFFFFFF 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)

25. Tick Timer Functions

This chapter details the functions for controlling the Tick Timer on the JN51xx 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 1: For guidance on using the tick timer functions in JN5148/JN5139 application code, refer to Chapter 9.

Note 2: On the JN5139 device, 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:

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.

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

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 Only)

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

Description

This function registers a user-defined callback function that will be called on a JN5139 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 is **vAHI_TickTimerInit()**.

Parameters

prTickTimerCallback	Pointer to callback function to be registered

Returns

None

26. Watchdog Timer Functions (JN5148 Only)

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

Note: For information on the watchdog timer and guidance on using the watchdog timer functions in JN5148 application code, refer to Chapter 10.

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

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 $1 \leq u8Prescale \leq 12$

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

27. Pulse Counter Functions (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. The two counters can be combined together to provide a single 32-bit counter, if desired.

Note: For information on the pulse counters and guidance on using the pulse counter functions in JN5148 application code, refer to Chapter 11.

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

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 passes 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 - that is, when the count reaches *(reference value + 1)*. 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 pass 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

28. Serial Interface (2-wire) Functions

This chapter details the functions for controlling the 2-wire Serial Interface (SI) on the JN51xx 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 the JN51xx microcontrollers and functions for controlling the SI master are described in Section 28.1.
- An SI slave is provided only on the JN5148 device and the functions for controlling the SI slave are described in Section 28.2.

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 JN5148/ JN5139 application code, refer to Chapter 12.

28.1 SI Master Functions

This section details the functions for controlling a 2-wire Serial Interface (SI) master on a JN51xx 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 28.2). 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:

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).

For more details of implementing a data transfer on the SI bus, refer to Section 12.1.

***Caution:** If interrupts are enabled, this function should not be called from the user-defined callback function registered via **vAHI_SiRegisterCallback()**.*

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 (should not normally be required as interrupt is cleared by the interrupt handler): E_AHI_SI_IRQ_ACK E_AHI_SI_NO_IRQ_ACK (normally the required setting)

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()**.

For more details of implementing a data transfer on the SI bus, refer to Section 12.1.

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

28.2 SI Slave Functions (JN5148 Only)

This section details the functions for controlling a 2-wire Serial Interface (SI) slave on the JN5148 microcontroller.

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

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^2C 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

29. SPI Master Functions

This chapter details the functions for controlling the Serial Peripheral Interface (SPI) on the JN51xx 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.

Note 1: For information on the SPI master and guidance on using the SPI master functions in application code, refer to Chapter 13.

Note 2: SPI slave functions are detailed in Chapter 30.

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

vAHI_SpiConfigure

```
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.

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 device, 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 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 device - 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 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 device - 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 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 device - 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

30. Intelligent Peripheral (SPI Slave) Functions

This chapter details the functions for controlling the Intelligent Peripheral (IP) interface of the JN51xx microcontrollers. The IP interface is a SPI (Serial Peripheral Interface) slave, designed to allow message passing and data transfer.

Note 1: For information on the IP interface (SPI slave) and guidance on using the IP functions in JN5148/ JN5139 application code, refer to Chapter 14.

Note 2: SPI master functions are detailed in Chapter 29.

Note 3: For more details of the data message format, refer to the data sheet for your microcontroller.

The IP (SPI slave) functions are listed below, along with their page references:

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 is supported - SPI mode 0. At both ends of the data link, the data to be transmitted is changed on a negative clock edge and received data is sampled on a positive clock edge. Therefore, the parameters bTxEdge and bRxEdge must be set accordingly (both to 0).*

Parameters

bTxEdge	Clock edge that transmit data is changed on (see Caution). Always set to 0, meaning that data is changed on a negative clock edge
bRxEdge	Clock edge that receive data is sampled on (see Caution). Always set to 0, meaning that data is sampled on a positive clock edge
bIntEn	Enable/disable Intelligent Peripheral interrupts: TRUE - enable interrupts FALSE - disable interrupts

Returns

None

vAHI_IpEnable (JN5139 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 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 is supported - SPI mode 0. At both ends of the data link, the data to be transmitted is changed on a negative clock edge and received data is sampled on a positive clock edge. Therefore, the parameters bTxEdge and bRxEdge must be set accordingly (both to 0).*

Parameters

bTxEdge	Clock edge that transmit data is changed on (see Caution). Always set to 0, meaning that data is changed on a negative clock edge
bRxEdge	Clock edge that receive data is sampled on (see Caution). Always set to 0, meaning that data is sampled on a positive clock 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 Version)

```
bool_t bAHI_IpSendData(uint8 u8Length,
                       uint8 *pau8Data);
```

Description

This function is used on the JN5139 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 Version)

```
bool_t bAHI_IpReadData(uint8 *pu8Length,
                       uint8 *pau8Data);
```

Description

This function is used on the JN5139 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

31. DAI Functions (JN5148 Only)

This chapter details the functions for controlling the 4-wire Digital Audio Interface (DAI) on the JN5148 microcontroller. This interface allows communication with external devices that support various digital audio interfaces such as CODECs.

Note 1: For information on the DAI and guidance on using the DAI functions in JN5148 application code, refer to Chapter 15.

Note 2: 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 32.

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

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^2S-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^2S.

Parameters

u8Mode	Transfer mode: 00: I^2S-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^2S-compatible transfer modes
bWsPolarity	WS polarity: 1: WS inverted 0: WS not inverted (as in I^2S)

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

32. Sample FIFO Functions (JN5148 Only)

This chapter details the functions for controlling and monitoring the Sample FIFO interface of the JN5148 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).

Note: For information on the Sample FIFO interface and guidance on using the Sample FIFO functions in JN5148 application code, refer to Chapter 16.

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

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

33. External Flash Memory Functions

This chapter describes functions for erasing and programming a sector of an external Flash memory device. JN51xx 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 M25P10A Flash device fitted to JN5139 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 Section 17.1 for the different Flash devices).

Note 1: 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 (thus, starting in Sector 0).

Note 2: For guidance on using the Flash memory functions in JN5148/JN5139 application code, refer to Chapter 17.

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

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.

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 (JN5139 Only)

```
bool_t bAHI_FlashErase(void);
```

Description

This function erases the 32-KB sector of Flash memory (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 (JN5139 Only)

```
bool_t bAHI_FlashProgram(uint16 u16Addr,
                         uint16 u16Len,
                         uint8 *pu8Data);
```

Description

This function programs a block of Flash memory (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 (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 (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

```
void vAHI_FlashPowerDown(void);
```

Description

This function sends a ‘power down’ command to the Flash memory device attached to the JN5148/JN5139 device. This allows further power savings to be made when the 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:

- ST M25P10A - for JN5148 and JN5139 devices
- ST M25P40 - 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 JN5148/JN5139 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 JN5148/JN5139 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

```
void vAHI_FlashPowerUp(void);
```

Description

This function sends a ‘power up’ command to the Flash memory device attached to the JN5148/JN5139 device.

The following Flash devices are supported by this function:

- ST M25P10A - for JN5148 and JN5139 devices
- ST M25P40 - 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 JN5148/JN5139 device wakes from ‘sleep without memory held’ if the Flash device was powered down using **vAHI_FlashPowerDown()** before the JN5148/JN5139 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 JN5148/JN5139 wakes from sleep and therefore there is no need to call **vAHI_FlashPowerUp()**.

Parameters

None

Returns

None

Part III: 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. The full list of peripheral interrupt sources and the corresponding callback registration functions is provided in the table below.

Interrupt Source	Callback Registration Function
System Controller **	**vAHI_SysCtrlRegisterCallback()**
Analogue Peripherals (ADC)	**vAHI_APRegisterCallback()**
UART 0	**vAHI_Uart0RegisterCallback()**
UART 1	**vAHI_Uart1RegisterCallback()**
Timer 0	**vAHI_Timer0RegisterCallback()**
Timer 1	**vAHI_Timer1RegisterCallback()**
Timer 2 *	**vAHI_Timer2RegisterCallback()**
Tick Timer	**vAHI_TickTimerRegisterCallback() *** **vAHI_TickTimerInit()**
Serial Interface (2-wire)	**vAHI_SiRegisterCallback() *** **
SPI Master	**vAHI_SpiRegisterCallback()**
Intelligent Peripheral	**vAHI_IpRegisterCallback()**
Digital Audio Interface *	**vAHI_DaiRegisterCallback()**
Sample FIFO Interface *	**vAHI_FifoRegisterCallback()**
Encryption Engine	Refer to *AES Coprocessor API Reference Manual (JN-RM-2013)*

Table 8: Interrupt Sources and Callback Registration Functions

* JN5148 device only

** Includes DIO, comparator, wake timer, pulse counter, random number and brownout interrupts

*** Used for both SI master and SI slave interrupts

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.*

A.1 Callback Function Prototype and Parameters

The user-defined callback functions for all peripherals must be designed according to 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. The list of possible sources is given in Table 8. Enumerations for these sources are provided in the API and are detailed in Appendix B.1.
- *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 deatiled in Appendix B.2.

A.2 Callback Behaviour

Before invoking one of the callback functions, the API clears the source of the interrupt, so that there is no danger of the same interrupt causing the processor to enter a state of permanently trying to handle the same interrupt (due to a poorly written callback function). This also means that it is possible to have a NULL callback function.

The UARTs are the exception to this rule. When generating a 'receive data available' or 'time-out indication' interrupt, the UARTs will only clear the interrupt once the data has been read from the UART receive buffer. It is therefore vital that if UART interrupts are to be enabled, the callback function handles the 'receive data available' and 'time-out indication' interrupts by reading the data from the UART before returning.

Note: If the Application Queue API is being used, the above issue with the UART interrupts is handled by this API, so the application does not need to deal with it. For more information on this API, refer to the *Application Queue API Reference Manual (JN-RM-2025)*.

A.3 Handling Wake Interrupts

A JN51xx microcontroller can be woken from sleep by any of the following sources:

- Wake timer
- DIO
- Comparator
- Pulse counter (JN5148 only)

For the device to be woken by one of the above wake sources, interrupts must be enabled for that source at some point before the device goes to sleep.

Interrupts from all of the above sources are handled by the user-defined System Controller callback function which is registered using the function **vAHI_SysCtrlRegisterCallback()**. The callback function must be registered before the device goes to sleep. However, in the case of sleep without RAM held, the registered callback function will be lost during sleep and must therefore be re-registered on waking, as part of the cold start routine before the initialisation function **u32AHI_Init()** is called. If there are any System Controller interrupts pending, the call to **u32AHI_Init()** will result in the callback function being invoked and the interrupts being cleared. An interrupt bitmap *u32ItemBitmap* is passed into the callback function and the particular source of the interrupt (DIO, wake timer, etc) can be obtained from this bitmap by logical ANDing it with masks provided in the API and detailed in Appendix A.1.

Note: As an alternative, for some wake sources 'Status' functions are available which can be used to determine whether a particular source was responsible for a wake-up event (see below). However, if used, these functions must be called before any pending interrupts are cleared and therefore before **u32AHI_Init()** is called.

The above wake sources are outlined below.

Wake Timer

There are two wake timers (0 and 1) on the JN51xx microcontrollers. These timers run at a nominal 32 kHz and are able to operate during sleep periods. When a running wake timer expires during sleep, an interrupt can be generated which wakes the device. Control of the wake timers is described in Chapter 8.

Interrupts for a wake timer can be enabled using **vAHI_WakeTimerEnable()**. The timed period for a wake timer is set when the wake timer is started.

The function **u8AHI_WakeTimerFiredStatus()** is provided to indicate whether a particular wake timer has fired. If used to determine whether a wake timer caused a wake-up event, this function must be called before **u32AHI_Init()** - see Note above.

DIO

There are 21 DIO lines (0-20) on the JN51xx microcontrollers. The device can be woken from sleep on the change of state of any DIOs that have been configured as inputs and as wake sources. Control of the DIOs is described in Chapter 5.

The directions of the DIOs (input or output) are configured using the function **vAHI_DioSetDirection()**. Wake interrupts can then be enabled on DIO inputs using the function **vAHI_DioWakeEnable()**. The change of state (rising or falling edge) on which each DIO interrupt will be generated is configured using the function **vAHI_DioWakeEdge()**.

The function **u32AHI_DioWakeStatus()** is provided to indicate whether a DIO caused a wake-up event. If used, this function must be called before **u32AHI_Init()** - see Note above.

Comparator

There are two comparators (1 and 2) on the JN5148 and JN5139 devices. The device can be woken from sleep by a comparator interrupt when either of the following events occurs:

- The comparator's input voltage rises above the reference voltage.
- The comparator's input voltage falls below the reference voltage.

Control of the comparators is described in Section 4.3.

Interrupts for a comparator are configured and enabled using the function **vAHI_ComparatorIntEnable()**.

A function **u8AHI_ComparatorWakeStatus()** is provided to indicate whether a comparator caused a wake-up event. If used, this function must be called before **u32AHI_Init()** - see Note above.

Pulse Counter (JN5148 Only)

There are two pulse counters (0 and 1) on the JN5148 device. These counters are able to run during sleep periods. When a running pulse counter reaches its reference count during sleep, an interrupt can be generated which wakes the device. Control of the pulse counters is described in Chapter 11.

Interrupts for a pulse counter can be enabled when the pulse counter is configured using the function **bAHI_PulseCounterConfigure()**.

B. Interrupt Enumerations and Masks

This appendix details the enumerations and masks used in the parameters of the interrupt callback function described in Appendix A.1.

B.1 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 9 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 9: u32DeviceId Enumerations

* JN5148 device only

** Used for both SI master and SI slave interrupts

B.2 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 10: 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 11: 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 12: Timers (identical for all timers)

Mask (Bit)	Description
0	Single source for Tick-timer interrupt, therefore returns 1 every time

Table 13: 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 14: Serial Interface (2-wire) Master

Mask (Bit)	Description
E_AHI_SIS_ERROR_MASK (4)	I^2C 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 15: Serial Interface (2-wire) Slave

Mask (Bit)	Description
E_AHI_SPIM_TX_MASK (0)	Transfer has completed

Table 16: SPI Master

Mask (Bit)	Description
E_AHI_DAI_INT_MASK (0)	End of data transfer via the DAI

Table 17: 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 18: 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 19: 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 20: UART (identical for both UARTs)

Table 20 lists the UART interrupts from highest priority to lowest priority.

Revision History

Version	Date	Comments
1.0	16-July-2010	First release
2.0	24-Nov-2010	Information incorporated from former *Integrated Peripherals API Reference Manual (JN-RM-2001)*

Important Notice

Jennic reserves the right to make corrections, modifications, enhancements, improvements and other changes to its products and services at any time, and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders, and should verify that such information is current and complete. All products are sold subject to Jennic's terms and conditions of sale, supplied at the time of order acknowledgment. Information relating to device applications, and the like, is intended as suggestion only and may be superseded by updates. It is the customer's responsibility to ensure that their application meets their own specifications. Jennic makes no representation and gives no warranty relating to advice, support or customer product design.

Jennic assumes no responsibility or liability for the use of any of its products, conveys no license or title under any patent, copyright or mask work rights to these products, and makes no representations or warranties that these products are free from patent, copyright or mask work infringement, unless otherwise specified.

Jennic products are not intended for use in life support systems/appliances or any systems where product malfunction can reasonably be expected to result in personal injury, death, severe property damage or environmental damage. Jennic customers using or selling Jennic products for use in such applications do so at their own risk and agree to fully indemnify Jennic for any damages resulting from such use.

NXP Laboratories UK Ltd
(Formerly 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.nxp.com/jennic