Jennic

TECHNOLOGY FOR A CHANGING WORLD

Integrated Peripherals API User Guide

JN-UG-3066
Revision 1.0
16 July 2010

Contents

About this Manual

This manual describes the use of the Integrated Peripherals Application Programming Interface (API) to interact with the peripheral devices on the Jennic JN5139 and JN5148 wireless microcontrollers. The manual explains the basic operation of each peripheral and indicates how to use the relevant API functions to control the device from the application which runs on the CPU of the microcontroller.

Note 1: The C functions and associated resources of the Integrated Peripherals API are fully detailed in the *Integrated Peripherals API Reference Manual (JN-RM-2001)*. You should refer to the Reference Manual in conjunction with this User Guide throughout your application development.

Note 2: Not all of the peripherals described in this manual are featured on both the JN5139 and JN5148 wireless microcontrollers. Where a peripheral is restricted to one microcontroller, this will be indicated.

Organisation

This manual consists of 17 chapters and an appendix, including one chapter for each peripheral device on the Jennic wireless microcontrollers, as follows:

- Chapter 1 presents a functional overview of the 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.
- The Appendices provide useful 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
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-RM-2001	Integrated Peripherals API Reference Manual
JN-RM-2003	Board API Reference Manual
JN-DS-JN5139	JN5139 Data Sheet
JN-DS-JN5148	JN5148 Data Sheet

Feedback Address

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

Applications
Jennic Ltd
Furnival Street
Sheffield S1 4QT
United Kingdom

doc@jennic.com

1. Overview

This chapter introduces the Integrated Peripherals Application Programming Interface (API) that is used to interact with peripheral devices on the Jennic range of wireless microcontrollers.

1.1 JN5139/JN5148 Integrated Peripherals

The Jennic JN5139 and JN5148 wireless microcontrollers each feature a number of on-chip peripheral devices 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 every microcontroller - where a peripheral is featured only on a certain microcontroller, 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 JN5139 and Figure 2 for JN5148.

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

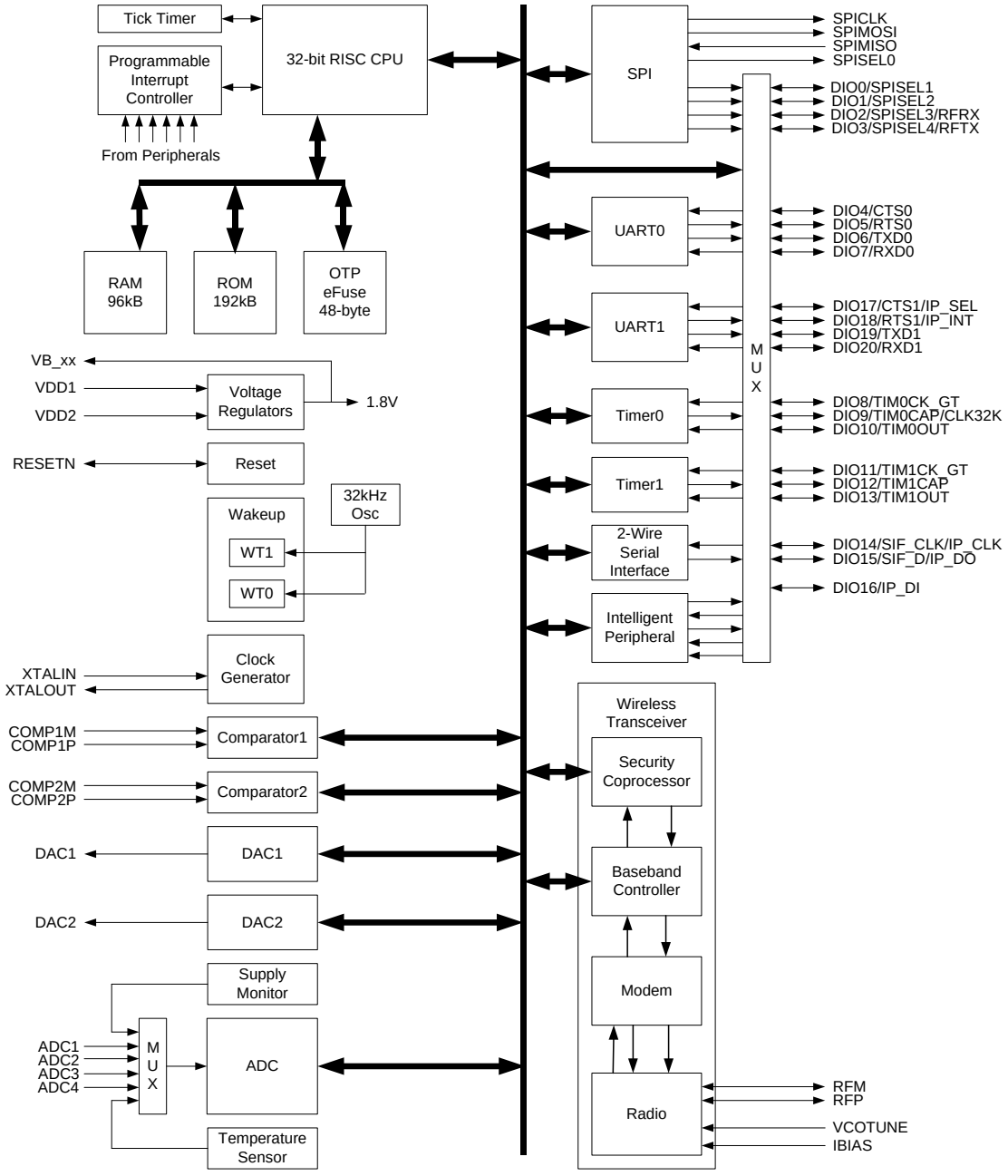


Figure 1: JN5139 Block Diagram

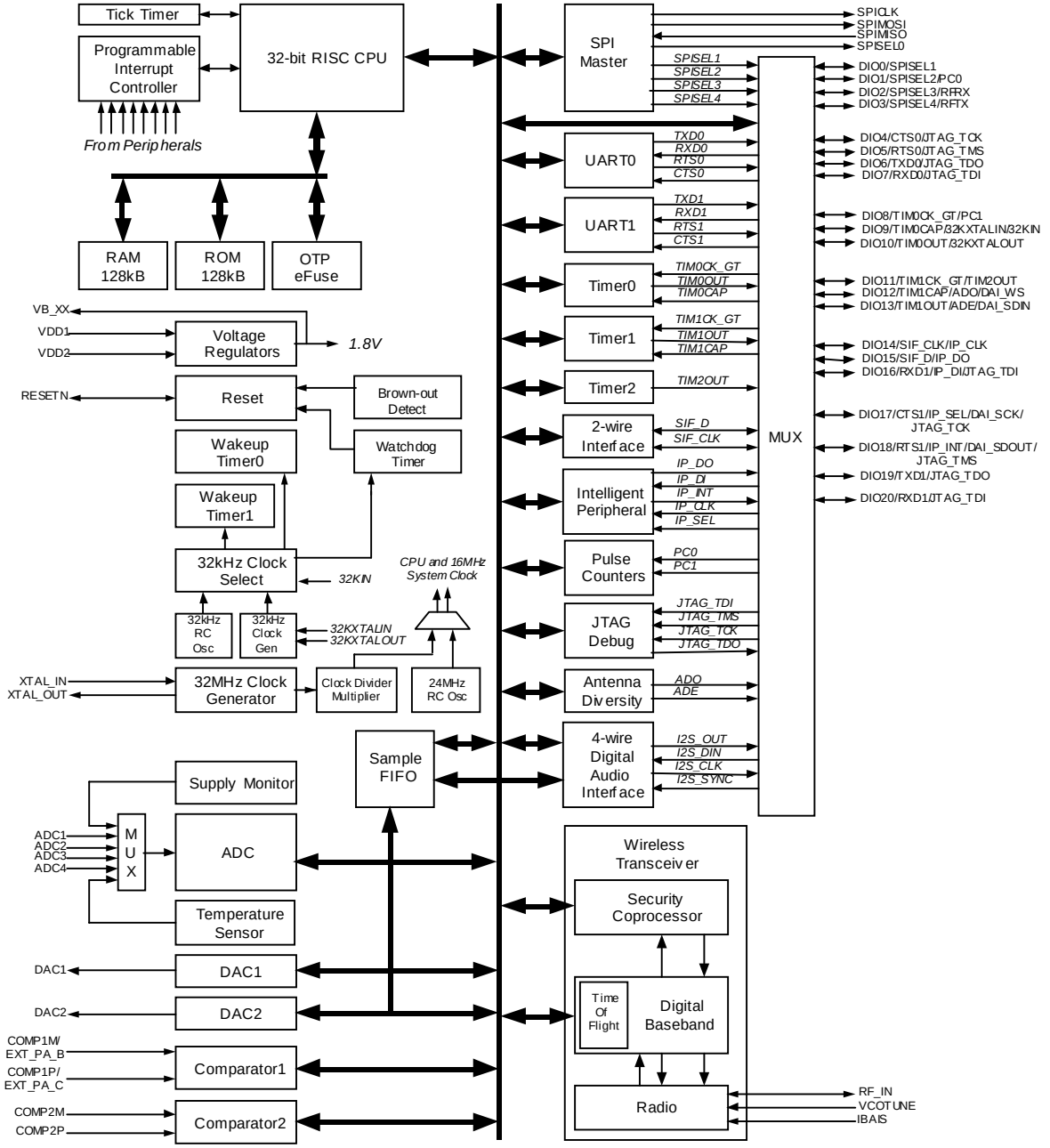


Figure 2: JN5148 Block Diagram

1.2 Integrated Peripherals API

The Integrated Peripherals API is a collection of C functions that can be incorporated in application code that runs on a Jennic wireless 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 Jennic SDK Libraries (JN-SW-4030 for JN5139, JN-SW-4040 for JN5148). 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 wireless microcontroller - this hardware is controlled by the wireless network protocol stack software (which may be an IEEE 802.15.4, ZigBee, Jenie/JenNet or 6LoWPAN/JenNet stack), and APIs for this purpose are provided with the appropriate stack software product.
- resources of the Jennic evaluation kit boards, such as sensors and display panels (although the buttons and LEDs on the Jennic evaluation kit boards are connected to the DIO pins of the wireless microcontroller) - a special function library, called the Board API, is provided by Jennic for this purpose and is described in the *Board API Reference Manual (JN-RM-2003)*.

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 which correspond to the particular peripherals that you wish to use in your application.

For full details of the API functions referenced in this manual, refer to the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

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 wireless 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 JN5139 or JN5148 wireless 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 JN5139 or JN5148 device can be varied, the exact power range depending on the device type and, more critically, the Jennic module type on which the device sits. As a general rule:

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

The transmission power can be set using the function **bAHI_PhyRadioSetPower()**. This function allows you to set the power to one of six (JN5139) or four (JN5148) possible levels in the power range - for details of these levels, refer to the function description in the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

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 Jennic JN5139 or JN5148 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 Jennic 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 wireless 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 Jennic wireless 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 Jennic wireless 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 Jennic's 802.15.4 Stack API, described in the *802.15.4 Stack API Reference Manual (JN-RM-2002)*:
  - 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 Jennic wireless 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 wireless 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 wireless 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 wireless 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 Jennic wireless 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 Jennic data sheet for your wireless 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: For details of the callback function prototype and the interrupt source bitmap, refer to the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

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 Jennic 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 Jennic wireless 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 Jennic data sheet for your wireless 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.

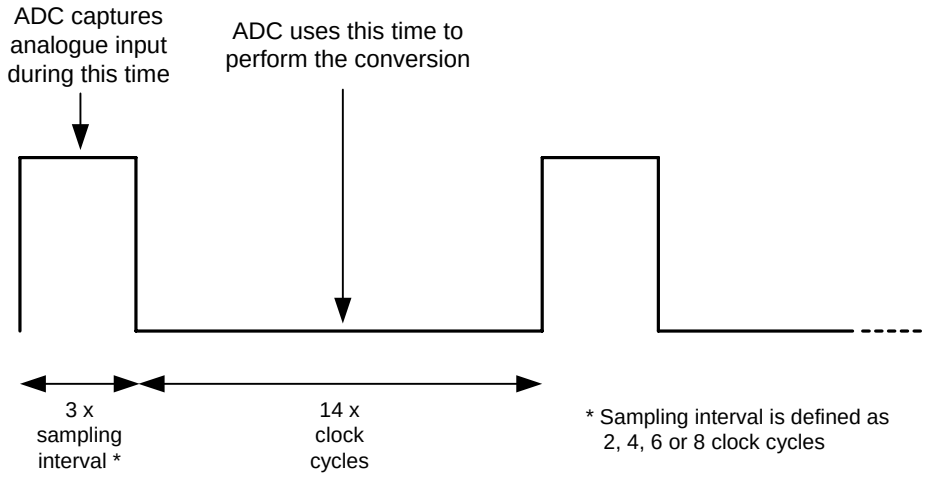


Figure 3: ADC Sampling

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 Jennic wireless 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 Jennic wireless 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.

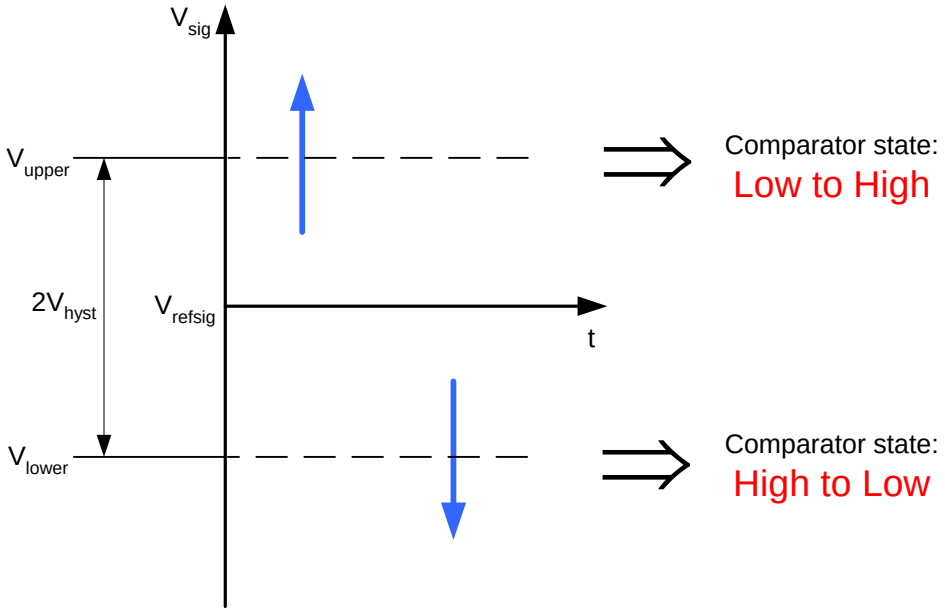


Figure 4: Upper and Lower Thresholds of Comparator

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 the *Integrated Peripherals API Reference Manual (JN-RM-2001)*. 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 Jennic wireless 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 wireless 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 wireless 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 wireless 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 Jennic wireless 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.

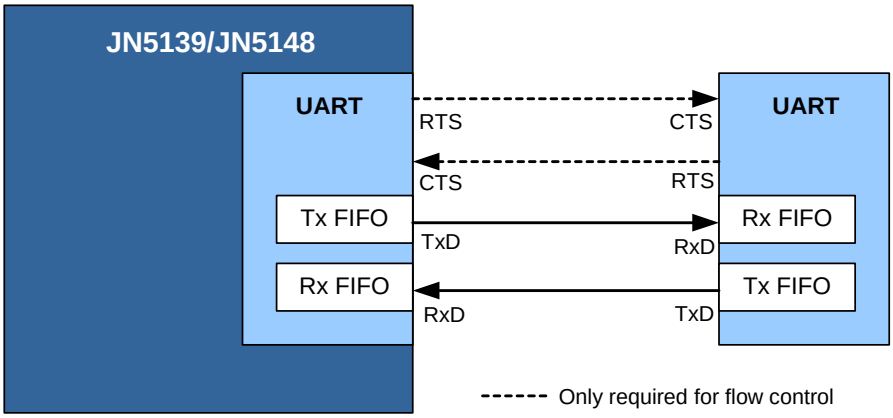


Figure 5: UART Connections

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.

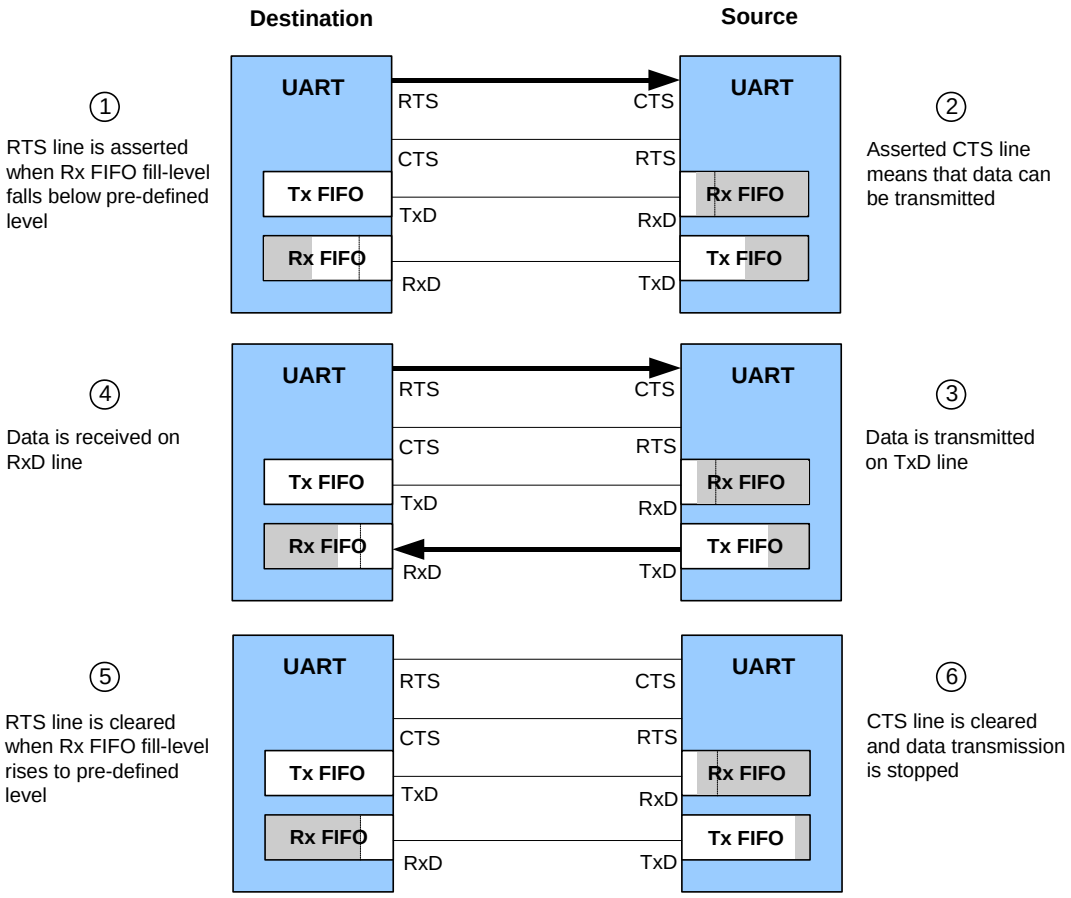


Figure 6: Example of UART Flow Control

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 the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

***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 the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

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.

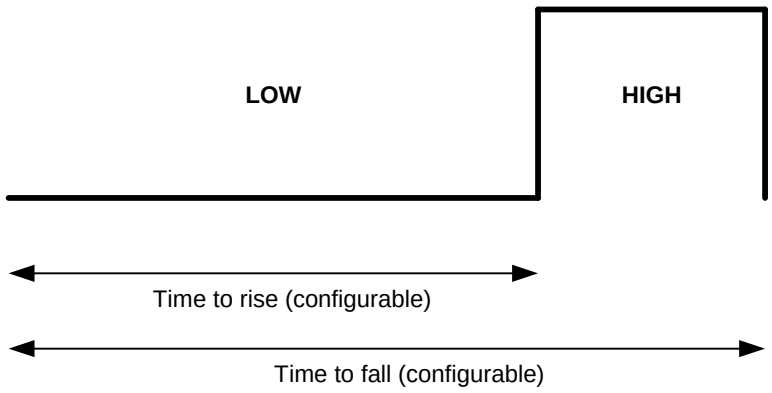


Figure 7: Timer Mode Signal

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 Jennic wireless 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.

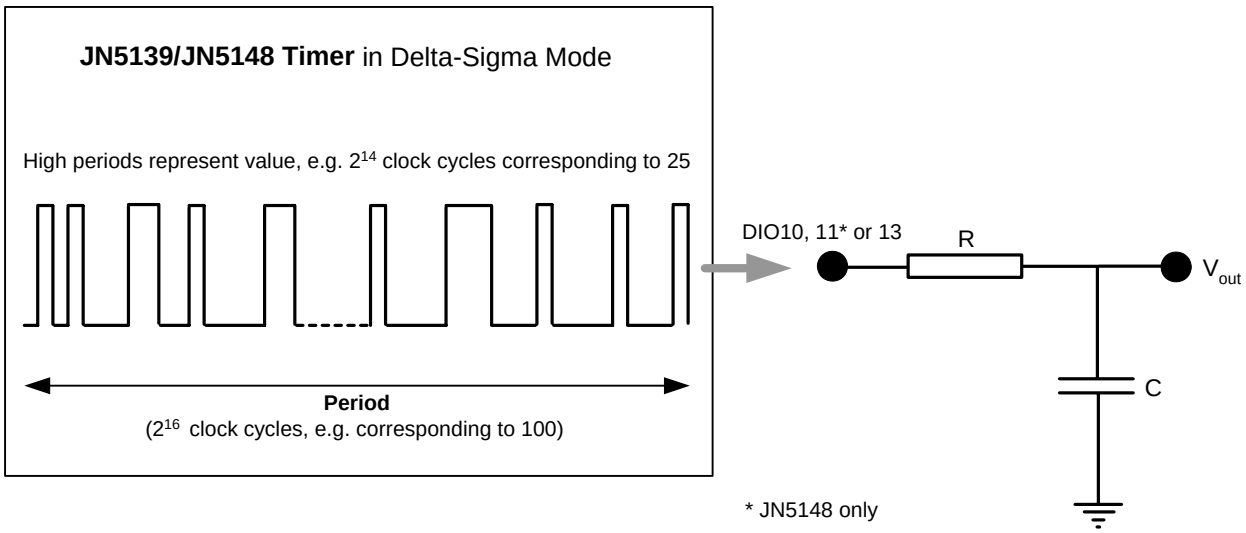


Figure 8: Delta-Sigma NRZ Mode Operation

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()**.

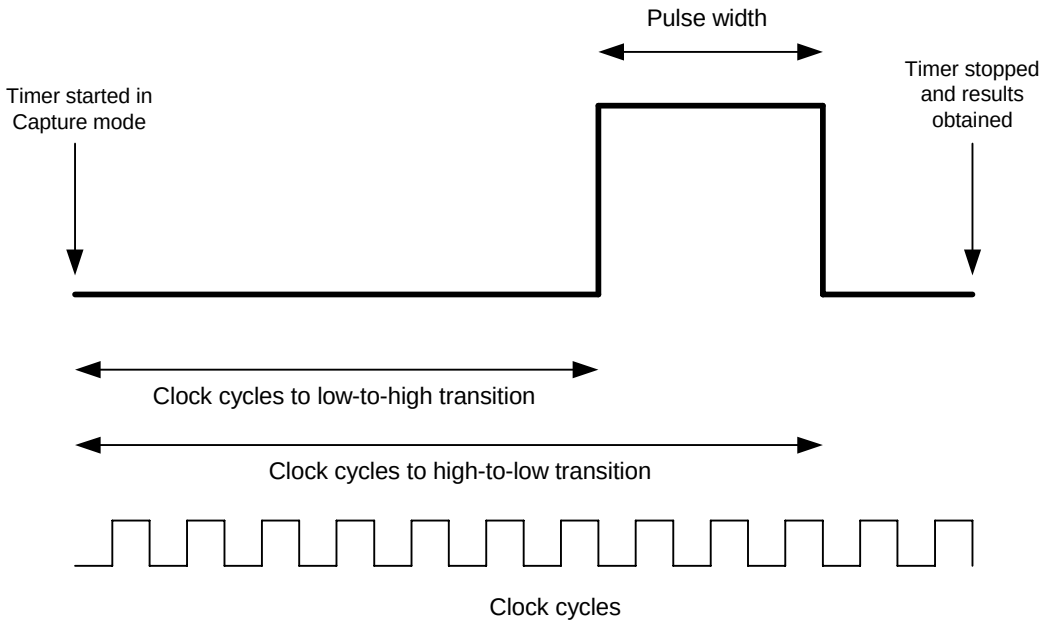


Figure 9: Capture Mode Operation

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. For details of the callback function prototype and the interrupt source bitmap, refer to the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

***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.*

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.

8. Wake Timers

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

The Jennic wireless 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 wireless 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 the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

***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 Jennic 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. 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 reaches 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 reached (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 reaches the pre-set reference value. 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 Jennic wireless 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 Jennic 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 all the Jennic wireless 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 wireless 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 the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

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 the *Integrated Peripherals API Reference Manual (JN-RM-2001)*. 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 the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

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 Jennic wireless 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. If enabled, the additional slave-select lines use pins DIO0-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 the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

***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 Jennic wireless 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 Jennic wireless 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 JN5139/JN5148 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.

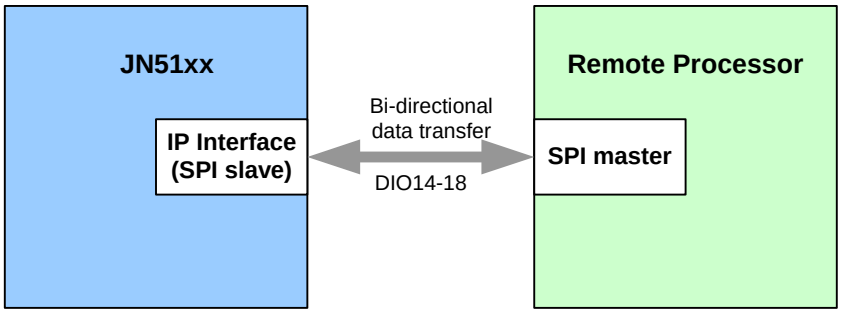


Figure 10: IP Interface as SPI Slave

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 JN5139/ JN5148 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 the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

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.

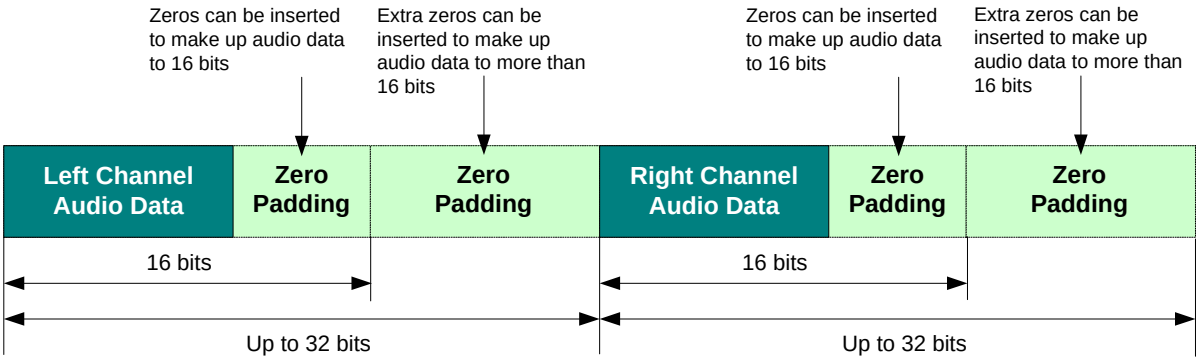


Figure 11: Format of Transferred Audio Data

15.1.3 Data Transfer Modes

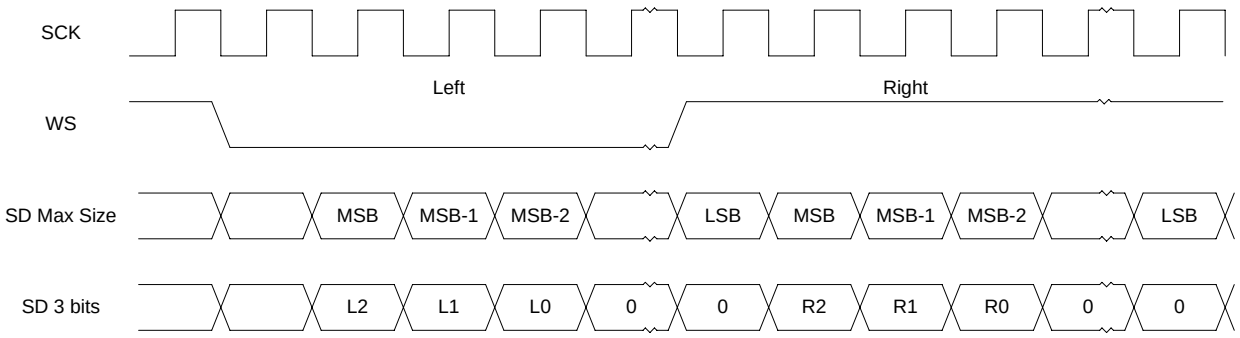
An audio data frame is always transferred from the DAI with left channel first and right channel second. Within a channel, the audio data is transferred starting with the most significant bit (MSB), although this bit may not be the first bit actually transferred (see modes below). The DAI will always 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.

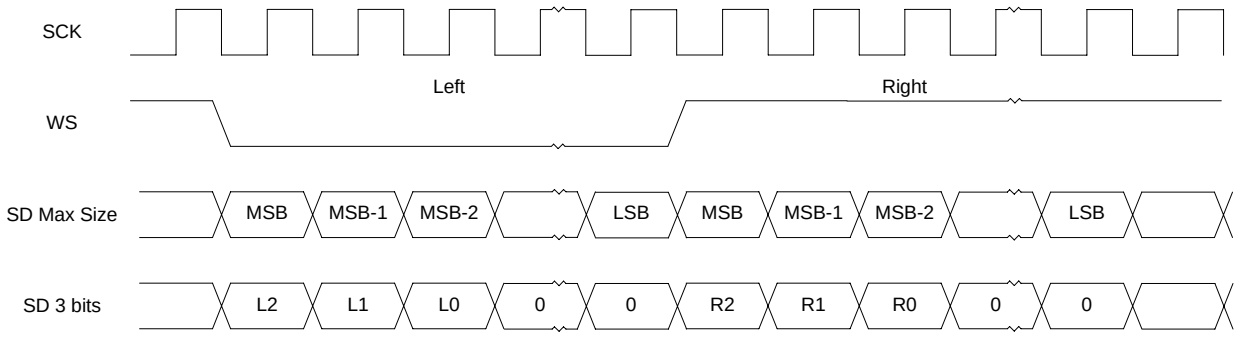
Figure 12: I²S Transfer Mode

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.

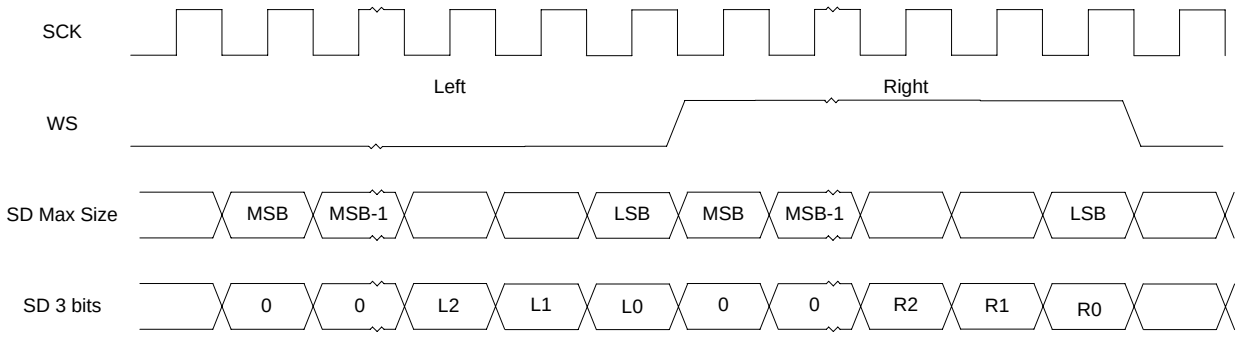
Figure 13: Left-justified Mode

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 the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

***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.

The Sample FIFO interface is illustrated in Figure 15 below.

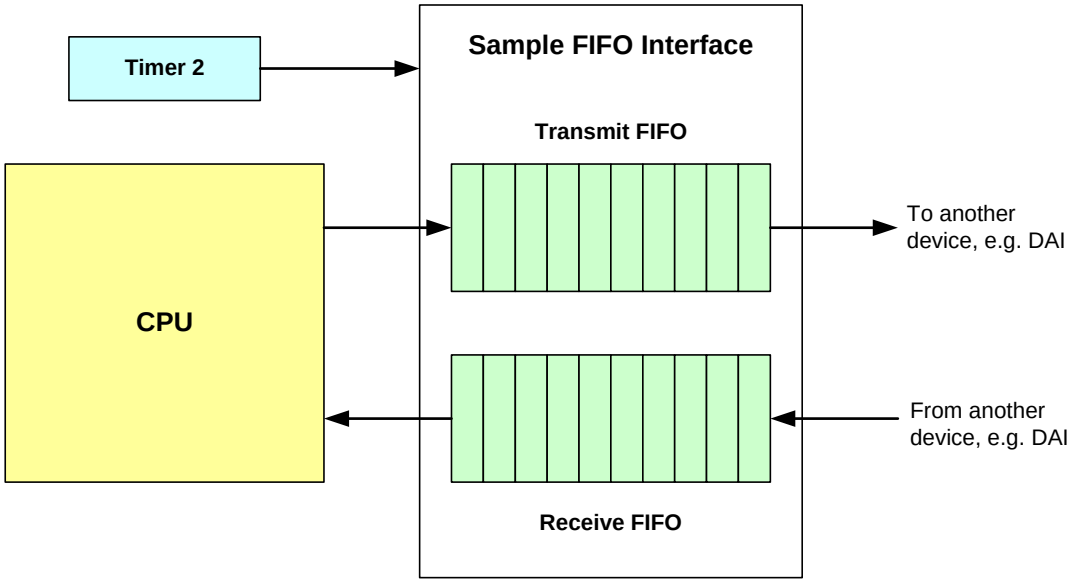


Figure 15: Sample FIFO Interface

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 the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

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 Jennic wireless 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

Jennic 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 Jennic wireless 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 any Jennic microcontroller 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 any Jennic microcontroller 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 any Jennic microcontroller 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 Jennic wireless 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 JN5139/JN5148 start-up and only power up the Flash device when required.

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.
- *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 used in the above parameters are detailed in the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

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 Jennic wireless 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 the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

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 Jennic wireless 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 Jennic wireless 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()**.

Revision History

Version	Date	Comments
1.0	16-July-2010	First release

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.

Jennic Ltd
Furnival Street
Sheffield
S1 4QT
United Kingdom

Tel: +44 (0)114 281 2655
Fax: +44 (0)114 281 2951
E-mail: info@jennic.com

For the contact details of your local Jennic office or distributor, refer to the Jennic web site:

www.Jennic.com
TECHNOLOGY FOR A CHANGING WORLD