JN517x Integrated Peripherals API User Guide

JN-UG-3118
Revision 1.2
22 December 2016

Contents

Preface

This manual describes the use of the JN517x Integrated Peripherals Application Programming Interface (API) to interact with the peripherals on a wireless microcontroller from the NXP JN517x family. The manual explains the basic operation of each peripheral and indicates how to use the relevant API functions to control the peripheral from the application which runs on the JN517x device. The C functions and associated resources of the API are fully detailed.

Organisation

This manual is divided into three parts:

- Part I: Concept and Operational Information comprises 17 chapters:
  - Chapter 1 presents a functional overview of the JN517x 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 and comparator.
  - Chapter 5 describes use of the **DIO functions**, used to control the general-purpose digital input/output pins.
  - Chapter 6 describes use of the **UART functions**, used to control the 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**, used to control the watchdog that allows software lock-ups to be avoided.
  - Chapter 11 describes use of the **Pulse Counter functions**, used to control the two pulse counters.
  - Chapter 12 describes use of the **Infra-Red Transmitter functions**, used to control the infra-red transmission feature of Timer 2.
  - Chapter 13 describes use of the **I^2C Interface functions**, used to control a 2-wire I^2C master and I^2C slave.
  - Chapter 14 describes use of the **Serial Peripheral Interface (SPI) Master functions**, used to control the master interface to the SPI bus.

  - Chapter 15 describes use of the **Serial Peripheral Interface (SPI) Slave functions**, used to control the slave interface to the SPI bus.
  - Chapter 16 describes use of the **Flash Memory functions**, used to manage the Flash memory.
  - Chapter 17 describes use of the **EEPROM functions**, used to access the on-chip EEPROM device.
- Part II: Reference Information comprises 16 chapters:
  - Chapter 18 details the **General functions** of the API, including the API initialisation function.
  - Chapter 19 details the **System Controller functions**, including functions that configure the system clock and sleep operations.
  - Chapter 20 details the **Analogue Peripheral functions**, used to control the ADC and comparator.
  - Chapter 21 details the **DIO functions**, used to control the general-purpose digital input/output pins.
  - Chapter 22 details the **UART functions**, used to control the 16550-compatible UARTs.
  - Chapter 23 details the **Timer functions**, used to control the general-purpose timers.
  - Chapter 24 details the **Wake Timer functions**, used to control the wake timers that can be employed to time sleep periods.
  - Chapter 25 details the **Tick Timer functions**, used to control the high-precision hardware timer.
  - Chapter 26 details the **Watchdog Timer functions**, used to control the watchdog that allows software lock-ups to be avoided.
  - Chapter 27 details the **Pulse Counter functions**, used to control the two pulse counters.
  - Chapter 28 details the **Infra-Red Transmitter functions**, used to control infra-red transmission.
  - Chapter 29 details the **I^2C Interface functions**, used to control a 2-wire I^2C master and I^2C slave.
  - Chapter 30 details the **Serial Peripheral Interface (SPI) Master functions**, used to control the master interface to the SPI bus.
  - Chapter 31 details the **Serial Peripheral Interface (SPI) Slave functions**, used to control the slave interface to the SPI bus.
  - Chapter 32 details the **Flash Memory functions**, used to manage the Flash memory.
  - Chapter 33 details the **EEPROM functions**, used to access the on-chip EEPROM device.
- Part III: Appendices provides information on handling interrupts from the peripheral devices.

Conventions

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

Function parameters are represented in *italics* type.

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

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

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

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

Acronyms and Abbreviations

ADC	Analogue-to-Digital Converter
AES	Advanced Encryption Standard
AHI	Application Hardware Interface
API	Application Programming Interface
CPU	Central Processing Unit
CTS	Clear-To-Send
DAC	Digital-to-Analogue Converter
DAI	Digital Audio Interface
DIO	Digital Input/Output
EIRP	Equivalent Isotropically Radiated Power
FIFO	First In, First Out (queue)
GPIO	General Purpose Input/Output
I^2C	Inter-Integrated Circuit
JCU	JN51xx Core Utilities

LPRF	Low-Power Radio Frequency
MAC	Medium Access Control
NVM	Non-Volatile Memory
PDM	Persistent Data Manager
PWM	Pulse Width Modulation
RAM	Random Access Memory
RTS	Ready-To-Send
SI	Serial Interface
SDK	Software Developer's Kit
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver-Transmitter
VBO	Voltage Brownout

Related Documents

JN517X	JN517x Data Sheet

Support Resources

To access online support resources such as SDKs, Application Notes and User Guides, visit the Wireless Connectivity area of the NXP web site:

www.nxp.com/products/wireless-connectivity

All NXP resources referred to in this manual can be found at the above address, unless otherwise stated.

Trademarks

All trademarks are the property of their respective owners.

Part I:
Concept and Operational Information

1. Overview

This chapter introduces the JN517x Integrated Peripherals Application Programming Interface (API) that is used to interact with peripherals on a wireless microcontroller from the NXP JN517x family. The chips of this family have the same peripherals but different memory sizes:

- JN5179 (32KB RAM, 4KB EEPROM, 512KB Flash memory)
- JN5178 (32KB RAM, 4KB EEPROM, 256KB Flash memory)
- JN5174 (32KB RAM, 4KB EEPROM, 160KB Flash memory)

1.1 JN517x Integrated Peripherals

The JN517x microcontrollers each feature a number of on-chip peripherals that can be used by a user application which runs on the CPU of the microcontroller. These 'integrated peripherals' are listed below.

- System Controller
- Analogue Peripherals:
  - Analogue-to-Digital Converter (ADC)
  - Comparator
- Digital Inputs/Outputs (DIOs)
- Universal Asynchronous Receiver-Transmitters (UARTs)
- Timers
- Wake Timers
- Tick Timer
- Watchdog Timer
- Pulse Counters
- I^2C Interface (2-wire):
  - I^2C Master
  - I^2C Slave
- Serial Peripheral Interface (SPI):
  - SPI Master
  - SPI Slave
- Interface to external Flash memory

The above peripherals are illustrated in Figure 1.

For hardware details of these peripherals, refer to the JN517x Data Sheet.

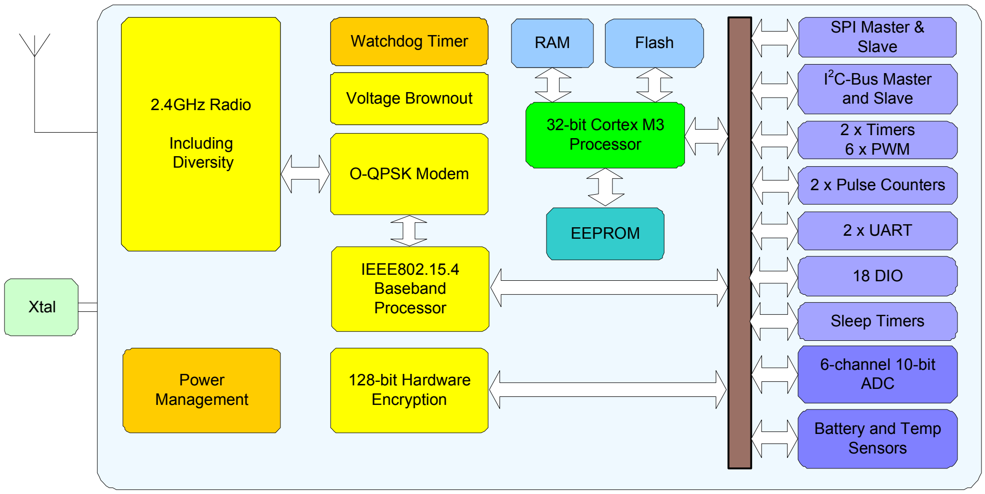


Figure 1: JN517x Block Diagram

1.2 JN517x Integrated Peripherals API

The JN517x Integrated Peripherals API is a collection of C functions that can be incorporated in application code that runs on a JN517x 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 NXP Software Developer's Kits (SDKs) for the JN517x devices. 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 JN517x 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 the:

- IEEE 802.15.4 Baseband Processor built into the JN517x device - this is controlled by the wireless network protocol stack software (which may be an IEEE 802.15.4, ZigBee, or Thread stack), and APIs for this purpose are provided with the appropriate stack software product.
- 128-bit AES Hardware Encryption core built into the JN517x device - this is controlled using the functions described in the *AES Coprocessor API Reference Manual (JN-RM-2013)*.

- EEPROM - this is controlled using the Persistent Data Manager provided as part of the ZigBee and Thread stacks. For further details, please refer to the *JN51xx Core Utilities User Guide (JN-UG-3116)*.
- resources of the JN517x evaluation kit boards, such as sensors and display panels (although the buttons and LEDs on the evaluation kit boards are connected to the DIO pins of the JN517x device) - a special function library, called the LPRF Board API, is provided by NXP for this purpose and is described in the *LPRF Board API Reference Manual (JN-RM-2003)*.

1.3 Using this Manual

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

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

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

2. General Functionality

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:

- API initialisation (Section 2.1)
- Configuration of the radio transmission power (Section 2.2)
- Use of Antenna Diversity (see Section 2.3)
- Use of the random number generator (Section 2.4)
- Accessing the JN517x internal Non-Volatile Memory (Section 2.5)
- Configuring a JN517x device to operate on a specific module (Section 2.6)

2.1 API Initialisation

Before calling any other function from the JN517x 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 JN517x microcontroller.

2.2 Radio Power

2.2.1 Transmission Power

The radio transmission power of a JN517x device can be varied. To set the transmission power, you can use the function **eAppApiPlmeSet()** from the NXP 802.15.4 Stack API (supplied in **AppApi.h** in the JN517x SDKs). The required function call is:

```
eAppApiPlmeSet(PHY_PIB_ATTR_TX_POWER, x);
```

where x is a 6-bit two's complement power level, corresponding to an input range of -32 to 31 dBm.

In practice, this value is mapped to an actual transmission level. For JN517x, it is mapped to the nearest of 26 levels in the range -32 to 10 dBm. Therefore, some positive input values will be truncated to 10 dBm.

2.2.2 Receive Power

The JN5169 device can receive radio signals with power of up to 10 dBm before the input is saturated. However, it is possible to configure the device to saturate at a reduced incoming signal power of 0 dBm, which has the advantage of drawing less current and prolonging battery life. This reduced maximum input level can be enabled using the function **vAHI_RadioSetReducedInputPower()**.

2.3 Antenna Diversity

The JN517x device provides an antenna diversity facility, allowing two antennas to be connected to the device. If this feature is implemented and the transmit and/or receive performance through the current antenna is deemed poor, a switch to the alternative antenna is automatically initiated.

If antenna diversity is to be used, two antennas must be connected to the JN517x device via a 2-state switch which is controlled by the device using a complementary pair of signals (ADO and ADE) output on DIO pins. On the JN5179 device, ADO is available for selection on DIO0, DIO4 or D01, and ADE is available on DIO1, DIO5 or DO0. The selection of which DIO/DO pin is used for each function is achieved through calls to the function **vAHI_SetDIOpinMultiplexValue ()**, allowing four signal combinations to be used. As an example, consider AD0 on pin DIO0 and ADE on pin DIO1. In one position (i.e. DIO0-1 = 10), the switch connects the RF_IN pin of the JN517x device to one antenna. In the other position (i.e. DIO0-1 = 01), the switch connects this pin to the other antenna. This connection is illustrated in Figure 2 below.

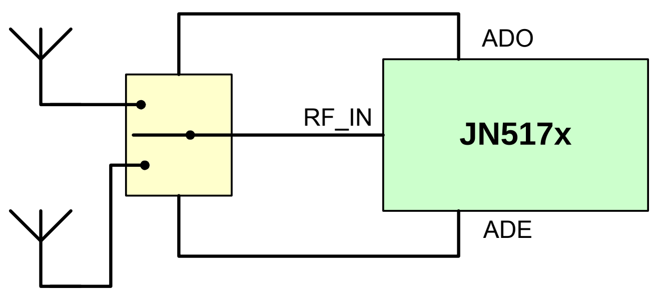


Figure 2: Connections for Antenna Diversity

Antenna diversity is enabled in the application by calling the function **vAHI_AntennaDiversityEnable()**. This function allows antenna diversity to be enabled individually for the transmit and receive paths (or for both paths). The operation of antenna diversity for the transmit and receive cases is outlined below:

- **Transmit:** For a transmission, the decision of whether to switch antennas is dependent on the use of IEEE 802.15.4 MAC acknowledgments. Once an IEEE 802.15.4 packet has been transmitted, the radio transceiver will enter receive mode and wait for an acknowledgment from the target node. If no acknowledgment is received, the device will retry the transmission on the alternative antenna (the number of retries is configurable in the IEEE 802.15.4 MAC). The selected antenna is switched for each subsequent retry.
- **Receive:** For reception, the JN517x device measures the received energy in the relevant radio channel every 40µs. The measured energy level is compared with a pre-set energy threshold. The JN517x device will switch the antenna if the measurement is below this threshold and all the following conditions hold:
  - The radio is not in the process of receiving a packet
  - A preamble symbol having a signal quality above a minimum specified threshold has not been detected in the last 40µs
  - The radio is not waiting for an acknowledgment from a previous transmission

The signal energy and signal quality thresholds can be set by the application using the function **vAHI_AntennaDiversityControl()**.

The current antenna diversity status can be obtained using the function **u8AHI_AntennaDiversityStatus()**. This function returns the antenna used for the last packet transmitted, the antenna used for the last packet received and the antenna that is currently selected.

The currently selected antenna can be manually switched by calling the function **vAHI_AntennaDiversitySwitch()**. Calling this function will generally not be required because it is expected that most applications will make use of the automatic transmit and/or receive antenna diversity control features that are enabled by calling **vAHI_AntennaDiversityEnable()**.

2.4 Random Number Generator

The JN517x devices feature 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 256µs.

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 32kHz clock domain (see Section 3.1) and will not operate properly if a high-precision external 32kHz 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. You may also generate random numbers in your application before switching to a high-precision external clock.

2.5 Accessing Internal NVM

The JN517x device contains a small block of Non-Volatile Memory (NVM) which is organised as four 32-bit words numbered 0, 1, 2 and 3. This memory can be used to preserve important data (e.g. counter values) at times when the JN517x RAM is not powered - for example, during periods of sleep without RAM held.

Two functions are provided to access this memory:

- **vAHI_WriteNVData()** can be used to write a 32-bit word of data to one of the four memory locations
- **u32AHI_ReadNVData()** can be used to read a 32-bit word of data from one of the four memory locations

> ***Caution:*** *The contents of this JN517x NVM are not maintained when the microcontroller is completely powered off. However, they are maintained through a device reset.*

2.6 Configuring the JN517x Device for a Specific Module

The JN517x devices are available on different types of module (a chip mounted on a mezzanine board), each with its own combination of features such as antenna type and transmission power. The function **vAHI_ModuleConfigure()** is provided to allow the application to configure the JN517x device for use with a particular module type - for details of the JN517x modules, refer to Section 2.6.2.

Depending on the module type, the function configures:

- Transmit power limits to ensure that the device remains within compliance limits
- CCA (Clear Channel Assessment) threshold to match the LNA (Low-Noise Amplifier) on the receive path
- DIO used to drive the PA (Power Amplifier) and LNA

For example, when using a JN517x standard-power module, it may be necessary to configure the device to operate within the compliance limits for a particular territory. Also, when using a JN517x high-power module, it is necessary to explicitly enable the high-power transmission capability of the module.

The above configuration can alternatively be performed using separate functions:

- **vAppApiSetComplianceLimits()** from the IEEE 802.15.4 Stack API
- **vAHI_SetDIOpinMultiplexValue()** from the JN517x Integrated Peripherals API

The function **vAHI_ModuleConfigure()** calls the above two functions.

It is important to note that the function **vAHI_ModuleConfigure()** is supplied as <u>source code only</u>, which can be customised (if required) and included in the JN517x application.

The code for **vAHI_ModuleConfigure()** is provided in the following two files:

- **Components/HardwareApi/Include/AHI_ModuleConfiguration.h**
- **Components/HardwareApi/Source/AHI_ModuleConfiguration.c**

Incorporating this function in your application is described in Section 2.6.1 below.

2.6.1 Adding the Function to an Application

To add the function **vAHI_ModuleConfigure()** to an application:

1. In the application source code, add:

```
#include "AHI_ModuleConfiguration.h"
```

2. Also in the source code, before initialising the stack add:

```
vAHI_ModuleConfigure(<Module configuration enumeration>);
```

3. In the application makefile, add:

Components/HardwareApi/Source/AHI_ModuleConfiguration.c

to the list of source files (or the equivalent **.o** file to the list of objects, depending on how the makefile works).

2.6.2 JN517x Module Type Enumerations

In the function **vAHI_ModuleConfigure()**, the module type is specified through the parameter *eModule* using an enumeration from the `teModule` enumerated type:

```
typedef enum
{
    E_MODULE_DEFAULT = 0,
#if (defined JENNIC_CHIP_FAMILY_JN516x) && !(defined JENNIC_CHIP_JN5169)
    E_MODULE_JN5168_001_M00_ETSI_FCC = 0x10,
    E_MODULE_JN5168_001_M03_ETSI_FCC,
    E_MODULE_JN5168_001_M05_ETSI,
    E_MODULE_JN5168_001_M06_FCC,
#elif (defined JENNIC_CHIP_JN5169)
    E_MODULE_JN5169_001_M00_ETSI = 0x20,
    E_MODULE_JN5169_001_M03_ETSI,
    E_MODULE_JN5169_001_M00_FCC,
    E_MODULE_JN5169_001_M03_FCC,
    E_MODULE_JN5169_001_M06_FCC,
#elif (defined JENNIC_CHIP_FAMILY_JN517x)
    E_MODULE_JN5179_001_M10_ETSI = 0x30,
    E_MODULE_JN5179_001_M13_ETSI,
    E_MODULE_JN5179_001_M10_FCC,
    E_MODULE_JN5179_001_M13_FCC,
    E_MODULE_JN5179_001_M16_FCC,
    E_MODULE_JN5179_001_M16_FCC_LNA_BYPASS
#endif
} teModule;
```

The enumeration names are of the form E_MODULE_xxx_yyy or E_MODULE_xxx_yyy_zzz, where:

- xxx is the hardware module name but with dashes changed to underscores (JN5168_001_M00, JN5179_001_M16, etc)
- yyy is the region:
  - ETSI for Europe
  - FCC for the USA
  - ETSI_FCC suitable for both Europe and the USA
- zzz is a module-specific configuration option (currently there is only one):
  - LNA_BYPASS - For the JN5179_001_M16 module only, the LNA on the receive path is bypassed and so a different CCA threshold value is required. Note that an IO input to the module must also be driven low to enable this mode, but this is outside the scope of the API function.

Only the following enumerations are relevant to the JN517x modules:

```
E_MODULE_JN5179_001_M10_ETSI
E_MODULE_JN5179_001_M13_ETSI
E_MODULE_JN5179_001_M10_FCC
E_MODULE_JN5179_001_M13_FCC
E_MODULE_JN5179_001_M16_FCC
E_MODULE_JN5179_001_M16_FCC_LNA_BYPASS
```

In addition, E_MODULE_DEFAULT is a generic enumeration which is equivalent to the most stringent compliance limits for a module without an external PA. For the JN5179 device, this is mapped to E_MODULE_JN5179_001_M10_FCC.

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)
- Supply voltage monitoring (Section 3.3)
- Chip reset (Section 3.4)
- Interrupts (Section 3.5)

3.1 Clock Management

The System Controller provides clocks to the JN517x microcontroller and is divided into four main blocks - a system clock domain, a peripheral clock domain, a CPU clock domain and a 32kHz clock domain.

System Clock Domain

The system clock is a high-speed reference clock from which the peripheral clock and CPU clock are derived when the chip is fully operational. The clock for this domain is sourced from one of the following:

- External 32MHz crystal oscillator
- Internal high-speed RC oscillator

The crystal oscillator is driven from a 32MHz external crystal connected to device pins 5 and 6 on the JN517x. The domain will produce a 32MHz system clock when sourced from the crystal oscillator.

The uncalibrated RC oscillator runs at 27MHz nominally, but can be calibrated to run at approximately 32MHz. The RC oscillator is mainly provided for a quick start-up following sleep, since the RC oscillator can start much more quickly than the crystal oscillator.

The radio transceiver and some peripherals should not be used when sourcing the system clock from the RC oscillator. System clock start-up and source selection is described in described in Section 3.1.1 and Section 3.1.2.

Peripheral Clock Domain

The peripheral clock is derived from the system clock and is used as the clock reference for the on-chip peripherals including the modem and baseband processor. The peripheral clock operates at half the system clock frequency - the peripheral clock runs at 16MHz when the system clock is sourced from the external 32MHz crystal oscillator. In the case of the Timers, these have the ability to run directly from the 32MHz clock

CPU Clock Domain

The CPU clock is a divided down version of the system clock and is used as the clock reference for the microprocessor and memory subsystem. The CPU clock frequency selection is described in Section 3.1.3.

32kHz Clock Domain

The 32kHz clock domain is mainly used during low-power sleep states (but also for the random number generator on the JN517x device - see Section 2.4). 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 (see Chapter 11). The wake timers are driven from the 32kHz domain. The 32kHz clock for this domain can be sourced from one of the following:

- Internal RC oscillator
- External crystal
- External clock module

For the JN517x device, the crystal oscillator is driven from an external 32kHz crystal connected to DIO7 and DIO8. If used, the external clock module is connected to DIO7.

Source clock selection for this domain is described in Section 3.1.5.

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

3.1.1 System Clock Start-up and Source Selection

As stated in the introduction to Section 3.1, there are two possible sources for the system clock on the JN517x device:

- Internal high-speed RC oscillator
- External crystal oscillator

where the crystal oscillator provides a more accurate clock than the RC oscillator.

Following a reset, the JN517x device takes its system clock from the internal high-speed RC oscillator. By default, an automatic switch to the external 32MHz crystal oscillator is performed once the crystal oscillator has stabilised (this can take up to 1ms). Application code is executed immediately following a reset.

Once the device and system clock are fully up and running, the system clock source can be changed using the function **vAHI_SelectClockSource()**. The identity of the current source clock can be obtained by calling the function **bAHI_GetClkSource()**.

> **Note:** If the external crystal oscillator is to be used as the source for the system clock, once the automatic switch to the crystal oscillator has occurred (`bAHI_GetClkSource() == FALSE`), the function **vAHI_OptimiseWaitStates()** should be called. This function optimises the wait states for the JN517x internal Flash memory and EEPROM according to the system clock frequency in order to minimise access times.

The RC Oscillator may be calibrated to improve its frequency accuracy by calling the function **bAHI_TrimHighSpeedRCOsc()**.

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

- Uncalibrated, the RC oscillator will produce a system clock frequency of 27MHz ±18% (or 32MHz ±5% if calibrated)
- The full system cannot be run while using the RC oscillator - it is possible to execute code but it is not possible to successfully transmit or receive radio signals. Also, the peripheral clock may not be sufficiently accurate to support certain peripheral functions, such as UART communication.

Therefore, while using the RC oscillator, use of the radio transceiver should not be attempted, and the JN517x peripherals should be used with special care.

3.1.2 System Clock Start-up Following Sleep

By default, following sleep, the JN517x device takes its system clock from the internal high-speed RC oscillator, but performs an automatic switch to the external 32MHz crystal oscillator once the crystal oscillator has stabilised (can take up to 1ms). Thus, application code is executed immediately following sleep.

It is possible to continue using the internal high-speed RC oscillator (without the automatic switch). In this case, before going to sleep, it is necessary to call the function **vAHI_EnableFastStartUp()** with the manual switch option selected - this cancels the automatic switch to the crystal oscillator.

3.1.3 CPU Clock Frequency Selection

A range of CPU clock frequencies are available on the JN517x device. By default, the source clock frequency is halved to produce the CPU clock. Thus:

- Using the external crystal oscillator, the 32MHz source frequency will produce a CPU clock frequency of 16MHz
- Using the uncalibrated internal high-speed RC oscillator, the 27MHz source frequency will produce a CPU clock frequency of 13.5MHz (±18%).

> **Note:** The frequency of the high-speed RC oscillator can be adjusted to a calibrated 32MHz by calling **bAHI_TrimHighSpeedRCOsc()**.

However, alternative CPU clock frequencies can be configured using the function **bAHI_SetClockRate()**. A division factor must be specified for dividing down the source clock to produce the CPU clock. The possible division factors are 1, 2, 4, 8, 16 and 32:

- For a source clock of 32MHz, the possible CPU clock frequencies are then 1, 2, 4, 8, 16 and 32 MHz
- For a source clock of 27MHz, the possible CPU clock frequencies are then 0.84, 1.17, 3.38, 6.75, 13.5 and 27 MHz.

3.1.4 System Clock Operation at High Temperatures

When the external crystal oscillator is operating at high temperatures, typically in excess of 85°C depending on the oscillator's characteristics, it will run fast. In this case, it may be necessary to call the function **vAHI_ClockXtalPull()** to decrease (pull) the frequency and maintain the frequency tolerance within the 40ppm limit specified by the IEEE 802.15.4 standard. This frequency pulling is achieved by increasing the crystal load capacitance in the oscillator tuning circuit. The required additional capacitance must be specified in the function call.

For a detailed description of frequency pulling, refer to the description of **vAHI_ClockXtalPull()** on page 167.

3.1.5 32kHz Clock Selection

As stated in the introduction to Section 3.1, a choice of source for the 32kHz clock is available on the JN517x device. The selection of this source clock is detailed below.

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

The 32kHz clock can be optionally sourced from an external crystal or clock module. One of these external clock sources can be selected using the function **bAHI_Set32KhzClockMode()**. If required, this function should be called near the start of the application.

If selecting the external crystal oscillator using **bAHI_Set32KhzClockMode()**, this function must be called before Timer 0 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. This function starts the external crystal, which can take up to 1 second to stabilise, and the function waits for the crystal to become ready before returning.

Alternatively, if the external crystal oscillator is required, the function **vAHI_Init32KhzXtal()** can be called to start the crystal and switch to it immediately. This function returns straight away but the clock will take up to 1 second to stabilise. While waiting for the crystal to become stable, the application can perform other processing or put the JN517x device into sleep mode - in the case of sleep, the application should typically set a wake timer to wake the device after 1 second.

If selecting the external clock module (RC circuit), the accuracy of the clock frequency produced can be chosen by setting the current consumption of the circuit using the function **vAHI_Trim32KhzRC()**.

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

- The external clock module must be supplied on DIO7. You must first disable the pull-up on DIO7 using the function **vAHI_DioSetPullup()**.
- The external crystal oscillator must be attached on DIO7 and DIO8. The pull-ups on DIO7 and DIO8 are disabled automatically by **vAHI_Init32KhzXtal()**.

Note that there is no need to explicitly configure DIO7 or DIO8 as an input, as this is done automatically by **bAHI_Set32KhzClockMode()** and by **vAHI_Init32KhzXtal()**.

3.2 Power Management

This section describes how to control the power to a JN517x 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 JN517x microcontroller has a number of power domains, as follows:

- **Digital Logic domain:** This domain supplies the CPU and digital peripherals as well as the wireless transceiver (including encryption coprocessor and baseband controller). The clock from this domain to the wireless transceiver 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. The domain 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, comparator and 32kHz oscillators. The domain is driven from the external supply (battery) and is always powered. However, the wake timers and 32kHz oscillators may be powered or unpowered during sleep.

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 Wireless Transceiver Clock

The clock to the wireless transceiver can be enabled/disabled using the function **vAHI_ProtocolPower()**. However, disabling this clock outside of a reset or sleep cycle must be done with caution. The following points should be noted:

- Disabling this clock leaves the clock powered but disabled (gated).
- Disabling the 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 the clock, the MAC settings must be restored from the saved settings. You can save and restore the MAC settings using functions of the 802.15.4 Stack API, described in the *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*:
  - To save the MAC settings, use the function **vAppApiSaveMacSettings()**.
  - To restore the saved MAC settings, use the function **vAppApiRestoreMacSettings() -** the clock is automatically re-enabled, since this function calls **vAHI_ProtocolPower()**.
- Do not call **vAHI_ProtocolPower()** to disable the clock while the IEEE 802.15.4 MAC layer is active, otherwise the microcontroller may freeze.
- While the 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 JN517x microcontroller is 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. The on-chip RAM, the 32kHz oscillator, the comparator and the pulse counter 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 32kHz oscillator are to be powered off. The significance of the 32kHz oscillator and RAM during sleep is outlined below:

- **32kHz Oscillator:** The 32kHz oscillator (internal RC, external clock or external crystal) can, in theory, be either left running or stopped for the duration of sleep. However, 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. Also, if an external source is used for this oscillator, it is not recommended that the oscillator is stopped on entering sleep mode.

> **Note:** If the pulse counter is to be run with debounce while the device is asleep, the 32kHz oscillator must be left running - see Chapter 11.

- **On-chip RAM:** Power to on-chip RAM can be either maintained or removed during sleep. The application program, stack context data and application data are all held in on-chip RAM while the microcontroller is fully active, but are lost if the power to RAM is switched off.
  - If the power to RAM is removed during sleep, the application is re-loaded into RAM from on-chip Flash memory on exiting sleep mode. Stack context and application data may also be re-loaded by the application, if they were saved to the on-chip EEPROM 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 into 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 system and 32kHz clock domains are powered down. In addition, any external Flash memory 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 an external Flash memory device using the function **vAHI_FlashPowerDown()**, which must be called before **vAHI_Sleep()**, provided you are using a compatible Flash device. For full details, refer to Section 16.4.

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

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

The device can only be woken from Deep Sleep mode by its reset line being pulled low 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 any interrupt.

3.2.4 Power Status

The power status of the JN517x microcontroller can be obtained using the function **u16AHI_PowerStatus()**. This function returns a bitmap which indicates 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
- Watchdog timeout was responsible for the last device restart
- 32kHz clock is ready (e.g. following a reset or wake-up)
- Device has just come out of Deep Sleep mode (rather than a reset)

For further details of the bitmap, refer to the function descriptions in Chapter 19.

3.3 Supply Voltage Monitor (SVM)

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 JN517x microcontroller is equipped with a Supply Voltage Monitor (SVM) to detect the brownout condition. SVM can be configured and monitored through functions of the Integrated Peripherals API.

3.3.1 Configuring SVM

By default on the JN517x device, the SVM feature is automatically enabled and the brownout voltage is set to 2.0V. On detection of a brownout, the chip will be automatically reset.

The SVM settings can be changed from the default values by calling the function **vAHI_BrownOutConfigure()**, which allows the configuration of the following:

- **SVM enable/disable:** The SVM feature can be enabled/disabled - if the configuration function is called and SVM is required, the feature must be explicitly enabled in the function.
- **Brownout level:** The brownout voltage level can be set to one of the following values: 1.95V, 2.0V (default), 2.1V, 2.2V, 2.3V, 2.4V, 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 before the new settings take effect - this delay is up to 3.3µs.

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

3.3.2 Monitoring Voltage

Provided that SVM is enabled (see Section 3.3.1), the brownout status of the JN517x 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 SVM 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 JN517x 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. Thus, any external devices connected to this line may be affected.

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

3.5 System Controller Interrupts

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

- Comparator
- DIOs
- Wake Timers
- Pulse Counter
- Random Number Generator
- Brownout detector

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

The handling of interrupts from these sources must be incorporated in a user-defined callback function, registered using the function **vAHI_SysCtrlRegisterCallback()**.

The registered callback function is automatically invoked when an interrupt of the type E_AHI_DEVICE_SYSCTRL occurs. The exact source of the interrupt (from the peripherals listed above) can then be identified from a bitmap that is passed into the function. Note that the interrupt will be automatically cleared before the callback function is invoked.

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

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

4. Analogue Peripherals

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

The are two types of analogue peripheral on the JN517x microcontroller:

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

Analogue peripheral interrupts are described in Section 4.4. The operation of the ADC with the DMA engine is described in Section 4.2.

4.1 ADC

The JN517x microcontroller includes a 10-bit Analogue-to-Digital Converter (ADC). The ADC samples an analogue input signal to produce a digital representation of the input voltage. It samples the input voltage at one instant in time and holds this voltage (in a capacitor) while converting it to a 10-bit binary value - the total sample/convert duration is called the conversion time.

The ADC may sample periodically to produce a sequence of digital 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: When an ADC input which is shared with a DIO is used, the associated DIO should be configured as an input with the pull-up disabled (refer to Section 5.1.1 and Section 5.1.3).

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 peripheral clock, normally 16MHz (see Section 3.1 for system clock options), which is divided down. The target frequency is selected using **vAHI_ApConfigure()**. The recommended target frequency for the ADC is 500kHz.

- **Sampling interval and conversion time:**

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

  The time allowed to perform the subsequent conversion is 13 clock periods. Thus, the total time to sample and convert (the conversion time) is given by:

  [(3 x *sampling interval)* + 13] 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 to V_{ref} or 0 to $2V_{ref}$, which is selected using the **vAHI_AdcEnable()** function - see later.

- **Voltage regulator:**

  In order to minimise the amount of digital noise in the ADC, the device is powered from a 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 a number of multiplexed sources comprising external pins (shared with DIOs), an on-chip temperature sensor and an internal voltage monitor. Six external input pins are available on the JN517x device and four pins on all other JN517x devices. 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 to V_{ref} or 0 to $2V_{ref}$ (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:** 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)* + 13] x *clock period*, which is 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.

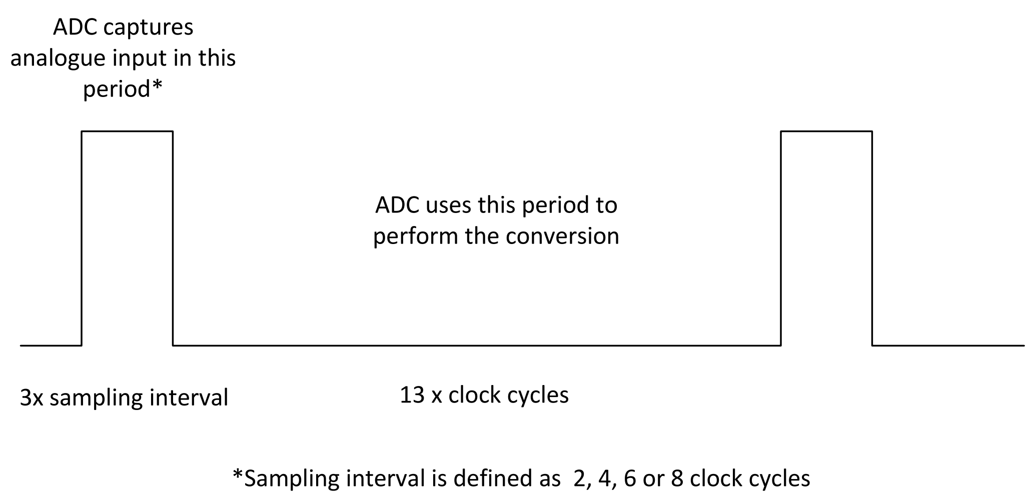


Figure 3: ADC Sampling

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: 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 result can be obtained using the function **u16AHI_AdcRead()**.

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)* + 13] 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 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

In accumulation mode, 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)* + 13] 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 ADC with DMA Engine (Sample Buffer Mode)

This section describes an operational mode of the ADC in which it is used in conjunction with the DMA (Direct Memory Access) engine on the JN517x device. In this mode:

- ADC 10-bit data samples are produced at regular intervals and transferred into a buffer in RAM as 16-bit samples, where this data transfer and storage is performed by the DMA engine independently of the CPU
- The CPU can perform other tasks while the data transfer and storage is being managed by the DMA engine - the CPU only needs to initiate the ADC conversions and deal with the results in the buffer (when an interrupt occurs)
- ADC sampling can be multiplexed between different analogue sources

This method of using the ADC is called ‘sample buffer mode’.

The ADC samples are produced at a configurable rate and are timed using the Analogue Peripheral Timer (APT), Timer8.

The application running on the CPU can service the buffer when the latter has collected sufficient data to cause an interrupt. The application must register a callback function to service this interrupt.

4.2.1 Preparing for Sample Buffer Mode

Before sample buffer mode can be enabled and started (see Section 4.2.2), the following preparations must be carried out:

- The function **vAHI_ApConfigure()** must be called to perform the initial configuration of the ADC (including clock frequency for conversion, sampling interval for conversion, reference voltage for input, use of voltage regulator and use of interrupts), as described for other ADC modes in Section 4.2.
- The Analogue Peripheral Timer must be set up and started to trigger the repeated conversions, as described in Chapter 7. Note the following:
  - There are no DIOs associated with this timer
  - Timer interrupts are not required and should be disabled for this timer when **vAHI_TimerEnable()** is called (see Section 7.2.2)
  - The timer must be configured and started in ‘Timer repeat’ mode using **vAHI_TimerStartRepeat()** (see Section 7.3.1)
- A user-defined callback function to handle the interrupts generated in sample buffer mode must be registered using **vAHI_APRegisterCallback()**, as described in Section 4.4 (the required interrupt mode is specified later when sample buffer mode is enabled and started - see Section 4.2.2).

4.2.2 Sample Buffer Mode Operation

Once sample buffer mode has been prepared as described in Section 4.2.1 (ADC configuration, timer started, callback function registered), operation in this mode can be further configured and started using **bAHI_AdcEnableSampleBuffer()**. The following configuration must be carried out in this function call:

- The Analogue Peripheral Timer must be specified as the timer to be used
- The input voltage range must be specified as either 0 to V_{ref} or 0 to $2V_{ref}$, where the reference voltage V_{ref} has been specified in **vAHI_ApConfigure()**
- A bitmap must be provided which specifies the analogue input sources that will be multiplexed in this ADC mode - the possible sources include external input pins, an on-chip temperature sensor and an internal voltage monitor
- The RAM buffer to receive the data must be fully specified, as follows:
    - A pointer to the start of the buffer
    - The size of the buffer (in 16-bit samples, up to a maximum of 2047)
    - Whether the buffer will wrap around to the start (when it becomes full)
- The DMA interrupt mode to be used must be specified as one of:
    - Interrupt when the buffer fills to its mid-point
    - Interrupt when the buffer is full
    - Interrupt when the buffer has wrapped around to its start

If operation in this mode is continuous (the buffer wraps around), it can be stopped using the function **vAHI_AdcDisableSampleBuffer()**.

Notes on various aspects of sample buffer mode operation (input multiplexing, buffer wrap and DMA interrupts) are provided below.

Input Multiplexing

Sample buffer mode allows a number of analogue inputs to be multiplexed. These inputs comprise external inputs ADC0-5, an on-chip temperature sensor and an internal voltage monitor. The required multiplexed inputs are specified through a bitmap in the call to **bAHI_AdcEnableSampleBuffer()**.

16-bit samples from all the selected inputs will be produced on each timer trigger. These samples will be produced (and stored in the buffer) in the following order:

1. External input ADC0
2. External input ADC1
3. External input ADC2
4. External input ADC3
5. External input ADC4
6. External input ADC5
7. Temperature sensor
8. Voltage monitor

Buffer Wrap

In the call to **bAHI_AdcEnableSampleBuffer()**, the RAM buffer can be configured to wrap around:

- If the buffer wrap option is enabled then when the buffer becomes full, data will continue to be written from the start of the buffer again. In this case, earlier data will be over-written and will be lost unless the buffer has been read by the application. The application can be alerted of a full buffer using the 'buffer full' interrupt (see DMA Interrupts below).
- If the buffer wrap option is not enabled then when the buffer becomes full, an overflow condition will exist on the production of the next data sample. In this case, no new data can be stored until the buffer is read by the application. The application can be alerted of a full buffer using the 'buffer full' interrupt and of an overflow situation using the 'buffer overflow' interrupt (see DMA Interrupts below).

In both of the above cases, the application should ensure that data is promptly read from the buffer in order to avoid losing data.

DMA Interrupts

DMA interrupts are used to notify the application of the status of the RAM buffer. These interrupts are as follows:

- **Buffer half-full:** The buffer has been half-filled with data
- **Buffer full:** The buffer has been completely filled with data:
    - If the buffer wrap option is enabled then the buffer will wrap around
    - If the buffer wrap option is disabled then the buffer will overflow if not read
- **Buffer overflow:** The buffer has been completely filled with data and a new data sample is available which cannot be stored and will be lost (applicable when the buffer wrap option has been disabled)

One or more of the above interrupt conditions can be selected in the call to **bAHI_AdcEnableSampleBuffer()**. These interrupts must be serviced by the user-defined callback function registered using **vAHI_APRegisterCallback()**.

4.3 Comparator

The JN517x microcontroller includes one comparator, referred to as COMP1, which has two inputs: COMP1P on DIO17 and COMP1M on DIO18.

The comparator can be used to compare two analogue inputs. It 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). The comparator can be used as a basis for measuring the frequency of a time-varying analogue input when compared with a constant reference input.

One analogue input 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 can be supplied on the comparator's 'positive' pin (COMP1P) or 'negative' pin (COMP1M). It will be compared with a reference signal, which can be sourced internally or externally as follows:

- externally from the other comparator pin (COMP1P or COMP1M) that is not being used for the monitored input signal
- internally from the reference voltage V_{ref} (the source of V_{ref} is selected using the function **vAHI_ApConfigure()**)

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

> **Note 1:** By default, the comparator is 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.
>
> **Note 3:** When a comparator pin is used, the associated DIO should be configured as an input with the pull-up disabled (refer to Section 5.1.1 and Section 5.1.3).

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.

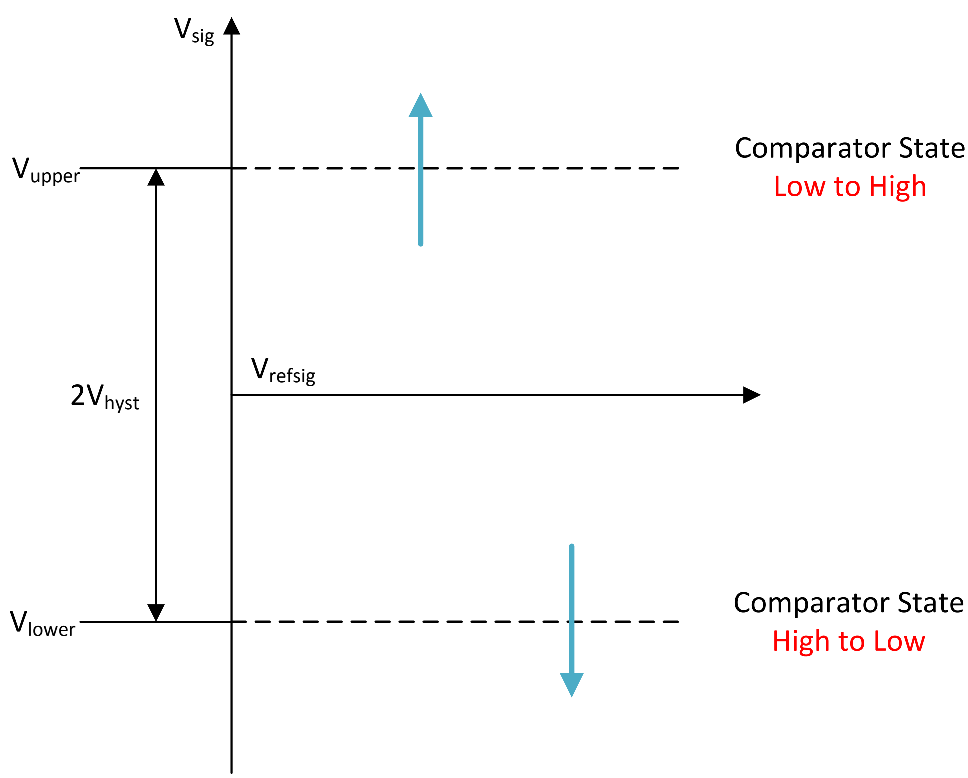


Figure 4: Upper and Lower Thresholds of Comparator

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 the 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 comparator allows 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 the ‘positive’ pin COMP1P, whichever is used during wake periods. The wake-up interrupt status can be checked using the function **u8AHI_ComparatorWakeStatus()**.

4.3.2 Comparator Low-Power Mode

The comparator is able to operate in a low-power mode, in which it draws only 0.8µA of current, compared with 73µA when operating in standard-power mode. Comparator low-power mode can be enabled/disabled using the function **vAHI_ComparatorLowPowerMode()**.

When a comparator is configured and started using **vAHI_ComparatorEnable()**, it operates in standard-power mode. To operate the comparator in low-power mode, the function **vAHI_ComparatorLowPowerMode()** must then be called.

Low-power mode is beneficial in helping to minimise the current drawn by a device that employs energy harvesting. It is also automatically enabled during sleep in order 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 comparator generates System Controller interrupts). The analogue peripheral interrupts are enabled in the function **vAHI_ApConfigure()** and are handled by a user-defined callback function registered using the function **vAHI_APRegisterCallback()**. For details of the callback function prototype, refer to Appendix A.1. The interrupt is automatically cleared when the callback function is invoked.

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

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

- In single-shot and continuous modes, a ‘capture’ interrupt will be generated after each individual conversion.
- In accumulation mode, 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 JN517x microcontroller has 18 DIO lines, numbered 0 to 15, 17 and 18. Each pin can be individually configured as an input or output. However, the DIO pins are shared with the following on-chip peripherals/features:

- ADC
- Comparator
- UARTs
- Timers
- 2-wire Inter-Integrated Circuit (I^2C) bus
- Serial Peripheral Interface (SPI)
- Antenna Diversity
- Pulse Counter

A shared DIO is not available when the corresponding peripheral/feature is enabled. For details of the shared pins, refer to the table following the description of **vAHI_SetDIOpinMultiplexValue()** in Chapter 21 or to the JN517x Data Sheet.

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.

> **Note:** In addition to the DIOs, the JN517x device has two digital outputs (DO0 and DO1). The configuration of these outputs is described in Section 5.3.

5.1 Using the DIOs

This section describes how to use the Integrated Peripherals API functions to configure and access 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 0-18 for DIO0-18, excluding bit 16). 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.

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 Enabling and Setting DIO Pull-ups/Pull-downs

Each DIO has an associated pull-up resistor and pull-down resistor. The purpose of these resistors is to prevent the state of the pin from 'floating' when there is no external load connected. When selected, 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). If the pull-down is selected, the pin is tied to the low (off) state in the absence of an external load (or in the presence a weak external load).

The pull-ups/pull-downs for all the DIOs can be enabled and disabled using the function **vAHI_DioSetPullup()**. The direction of the resistor (i.e. pull-up or pull-down) is set using the function **vAHI_DioSetDirection()**.

From reset, pull-ups and pull-downs are enabled - the selection of pull-up and pull-down resistors from reset is shown in Table 1 below.

DIO	Direction
18	Pull-up
17	Pull-up
15	Pull-down
14	Pull-down
13	Pull-down
12	Pull-down
11	Pull-up
10	Pull-up
9	Pull-up
8	Pull-down
7	Pull-down
6	Pull-up
5	Pull-up
4	Pull-up
3	Pull-down
2	Pull-up
1	Pull-up
0	Pull-up

Table 1: DIO Default Resistor Direction

If a shared DIO is in use by an on-chip peripheral when **vAHI_DioSetPullup()** or **vAHI_DioSetDirection()** is called, the specified pull-up setting for the DIO will be applied immediately.

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.

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.

5.1.5 Reading the State of Pull-ups/Pull-downs

The states of the pull-ups and pull-downs DIOs can be obtained using the **u32AHI_DioReadPullupDirection()**. This function will return the pull-up/pull-down states of all the DIOs, irrespective of whether they have been configured as inputs or outputs, or are in use by peripherals. Noted that the return value should be used in conjunction with the value used in **vAHI_DioSetPullup()** in order to ensure that the overall status takes into account whether the reported pull-up/pull-down state has been enabled.

5.1.6 Configuring the Functions of DIOs

The DIOs can be operated as general purpose inputs or outputs, or they can be dedicated to the various internal peripherals available in the JN517x device. Multiplexing hardware is provided which allows signals from a particular peripheral to be directed to one of possibly several pins on the device - similarly, inputs intended for the peripherals which are connected to a device pin can be routed to the correct peripheral input signal. This routing is performed using the function **vAHI_SetDIOpinMultiplexValue()**. The multiplexing is independent of the direction-setting and pull-up/pull-down operations described in Section 5.1.3 and Section 5.1.5. Therefore, these need to be configured at the same time as the DIO function is set. The function **u32AHI_ReadDIOpinMultiplexValue()** is provided to allow the multiplex settings to be read by the application.

5.2 DIO Interrupts and Wake-up

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

***Caution:** Since the 'DIO interrupt' and 'DIO wake' functions access the same JN517x register bits, you must ensure that the two sets of functions do not conflict in your application code.*

5.2.1 DIO Interrupts

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

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

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

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

5.2.2 DIO Wake-up

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

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

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

Note 1: As an alternative to calling the function **u32AHI_DioWakeStatus()**, you can determine the wake interrupt source in the callback function registered via **vAHI_SysCtrlRegisterCallback()**.

Note 2: When waking from deep sleep, the function **u32AHI_DioWakeStatus()** will not indicate a DIO wake source because the device will have completed a full reset. When waking from sleep, this function may indicate more than one wake source if multiple DIO events occurred while the device was booting.

5.3 Configuring Digital Outputs (DOs)

The JN517x device has two pins, DO0 and DO1,that may be used as general-purpose digital outputs while the device is awake. The DO pins are shared with the SPI Master, SPI Slave and Antenna Diversity functions.

These pins can be configured as follows:

- **bAHI_DoEnableOutputs()** can be used to enable (or disable) the DO pins as general-purpose digital outputs - by default, the DO pins are disabled as general-purpose digital outputs at power-up
- **vAHI_DoSetDataOut()** can be used to set the output states of the DO pins to on or off, in any combination - by default, the output states are on at power-up
- **vAHI_DoSetPullup()** can be used to set the pull-up states of the DO pins to on or off, in any combination - by default, the pull-ups are enabled at power-up

The DO pins do not preserve their status through sleep and may not be used to wake the device from sleep. From reset, during sleep and on waking from sleep the DO pins revert to being disabled as general-purpose outputs with pull-ups enabled.

6. UARTs

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

The JN517x microcontroller has two UARTs, denoted UART0 and UART1, which can be independently enabled. These UARTs are 16550-compatible and can be used for the input/output of serial data at a programmable baud-rate of up to 4Mbps.

> **Note:** The UART operation described here assumes that the peripheral clock runs at 16MHz and is derived from an external crystal oscillator - see Section 3.1. A system clock derived from the internal high-speed RC oscillator will cause inaccurate baud rates.

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

If a UART just uses signals RxD and TxD, it is said to operate in 2-wire mode (see Section 6.2.1). If it uses all four of the above signals, it is said to operate in 4-wire mode and implements flow control (see Section 6.2.2). On the JN517x device:

- UART0 can operate in 4-wire mode, 2-wire mode (default) or 1-wire (transmit only) mode
- UART1 can operate in 2-wire mode (default) or in 1-wire (transmit only) mode

The default pins used for the above signals are shared with the DIOs, as follows:

Signal	DIO Assignment	
	Standard Pins	Alternative Pins
CTS0	DIO6	-
RTS0	DIO11	-
RXD0	DIO10	DIO13
TXD0	DIO9	DIO12
RXD1	DIO2	DIO6
TXD1	DIO3	DIO11

Table 2: DIOs Used for UART Signals

The UART signals can be moved from their standard positions to their alternative positions using the function **vAHI_SetDIOpinMultiplexValue()**.

6.2 UART Operation

The transmit and receive paths of a UART each have a 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

The FIFOs are contained in RAM and are defined by the application. The size of each FIFO can be from 16 bytes up to 2047 bytes.

On the local device, the CPU can write/read data to/from a FIFO either one byte or a block of data at a time. The two paths are independent, so transmission and reception occur independently. The movement of data between the FIFOs and the TxD/RxD lines is handled by a DMA (Direct Memory Access) engine, with no CPU involvement.

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

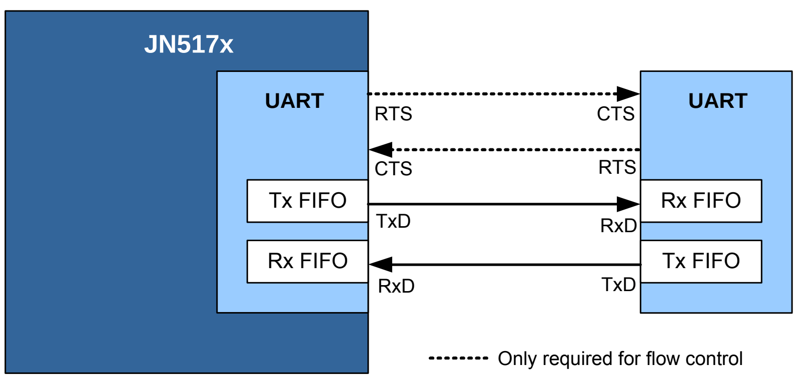


Figure 5: UART Connections

Both UARTs can operate in 2-wire mode and 1-wire mode, and UART0 can also operate in 4-wire mode. These modes 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) [UART0 Only]

In 4-wire mode, UART0 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.

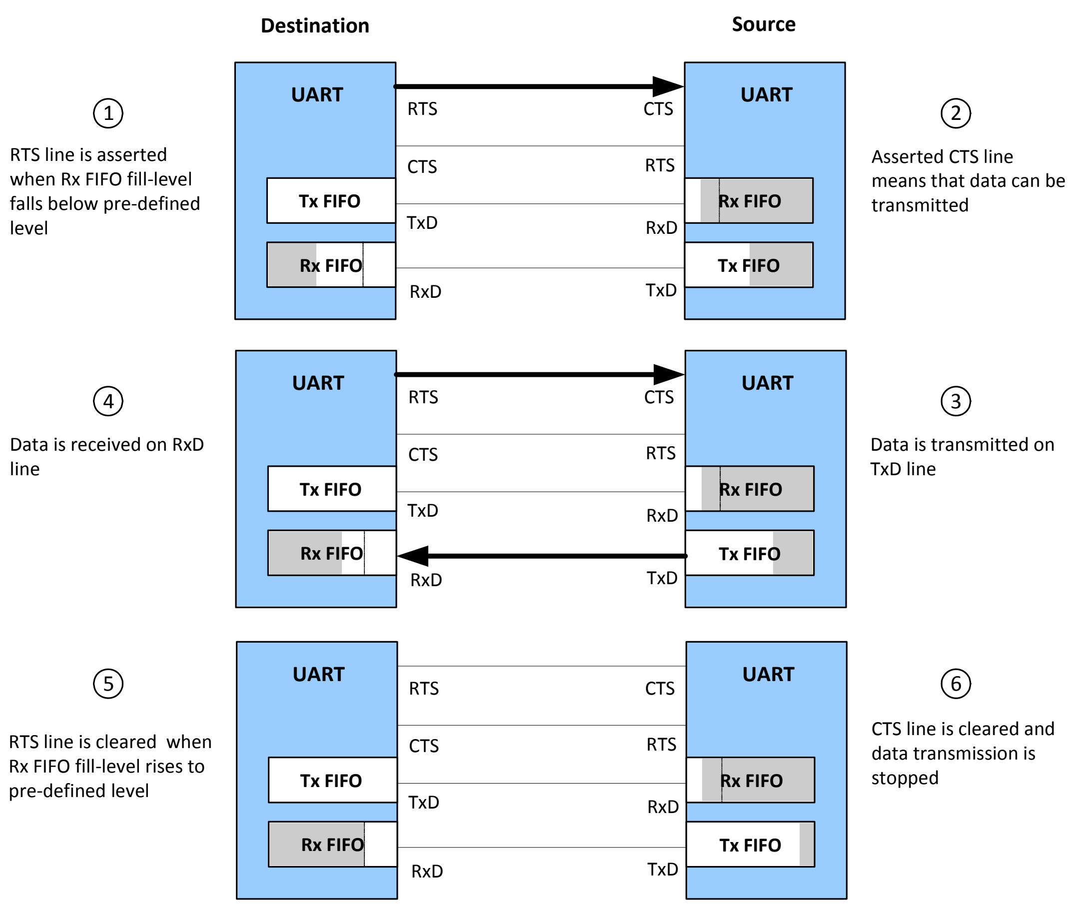


Figure 6: Example of UART Flow Control (UART0 Only)

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, 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.

6.2.3 1-Wire Mode [UART1 Only]

In 1-wire mode, UART1 uses the TxD line to transmit data unannounced, at the convenience of the sending device. In this mode, data is not received and the RxD line is unused (so the associated DIO pin is available for another purpose).

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 can be enabled using the function **bAHI_UartEnable()** or **bAHI_UartEnableNoneDIO()**. The choice of function depends on the DIOs to be used for the UART signals and the required mode of UART operation:

- **bAHI_UartEnable()** can be used when the standard DIOs (see Table 2 on page 63) are to be used for the UART signals (these DIOs are selected by the function). This function enables the UART in 2-wire mode. If used, it must be the first UART function called.
- **bAHI_UartEnableNoneDIO()** must be called when the DIOs to be used for the UART signals are selected separately (see below). This function enables the UART in 2-wire mode, by default, but allows UART0 to be used in 4-wire mode or either UART to be used in 1-wire mode. Note the following:
  - The DIOs for the UART signals are selected using the function **vAHI_SetDIOpinMultiplexValue()**. The possible DIO options are indicated in Table 2 on page 63. There must be a separate call to this function for each DIO pin used (so for 1-wire mode, it only needs to be called once, for the TxD signal).
  - To use UART0 in 4-wire mode, you will also need to call **vAHI_UartSetAutoFlowCtrl()** to enable Automatic Flow Control.

The functions **bAHI_UartEnable()** and **bAHI_UartEnableNoneDIO()** also allow the FIFO buffers for the UART transmit and receive paths to be configured. Each buffer is defined by the application as a section of RAM, and the start address and size (in bytes) of each buffer must be specified. The maximum possible buffer size is 2047 bytes and the minimum possible buffer size is 16 bytes.

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 1MHz frequency, given by:

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

- **vAHI_UartSetClocksPerBit()**

  This function can be used 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 16MHz peripheral 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 is 4Mbps (*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 UART0 in 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) in the case of UART0 operating 4-wire mode.

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 6.8.

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 1: For UART1, 2-wire mode is the default mode.

Note 2: In order to operate UART0 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 the function **vAHI_UartEnableNoneDIO()**.

6.4.1 Transmitting Data (2-wire Mode)

Data is transmitted via a UART by calling one of the following functions:

- **vAHI_UartWriteData()**: This function can be used to write a single byte of data to the Transmit FIFO. If used, this function may be called multiple times to queue data bytes for transmission.
- **u16AHI_UartBlockWriteData()**: This function is used to write a block of data bytes to the Transmit FIFO. The function will return the number of bytes that have been successfully written to the FIFO.

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 transfer of data from the FIFO to the TxD line is handled automatically by the DMA engine.

The following methods can be used to prompt the application to call the **vAHI_UartWriteData()** or **u16AHI_UartBlockWriteData()** function:

- The function **u16AHI_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).

6.4.2 Receiving Data (2-wire Mode)

Data is received on the RxD line as and when the source device sends it. The local transfer of data from the RxD line to the Receive FIFO is handled automatically by the DMA engine. The destination application can read data from the FIFO using one of the following functions:

- **u8AHI_UartReadData()**: This function can be used to read a single byte of data from the Receive FIFO.
- **u16AHI_UartBlockReadData()**: This function can be used to read a block of data bytes from the Receive FIFO.

The following methods can be used to prompt the application to call the **u8AHI_UartReadData()** or **u16AHI_UartBlockReadData()** function:

- The function **u16AHI_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 **u16AHI_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 (UART0 Only)

In 4-wire mode, UART0 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. The implementation of manual flow control is described below for transmission and reception separately, and then automatic flow control is described.

Note 1: 4-wire mode is the default mode on UART0. Therefore, the UART will automatically have control of the DIOs used for the RTS and CTS lines as soon as **vAHI_UartEnable()** is called.

Note 2: 4-wire mode is not available on UART1.

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, one of the following functions can be called:

- **vAHI_UartWriteData()**: This function can be used to write a single byte of data to the Transmit FIFO. If used, this function may be called multiple times to queue data bytes for transmission.
- **u16AHI_UartBlockReadData()**: This function can be used to read a block of data bytes from the Receive FIFO. The function will return the number of bytes that have been successfully written to the FIFO.

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 transfer of data from the FIFO to the TxD line is handled automatically by the DMA engine.

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 **u16AHI_UartReadTxFifoLevel()** or **u8AHI_UartReadLineStatus()**.

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 **u16AHI_UartReadRxFifoLevel()** 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()** or **vAHI_UartSetControl()**.

The source device may then send data, which is received on the RxD line on the destination device. The local transfer of data from the RxD line to the Receive FIFO is handled automatically by the DMA engine. The received data can be read from the Receive FIFO using one of the following functions:

- **u8AHI_UartReadData()**: This function can be used to read a single byte of data from the Receive FIFO.
- **u16AHI_UartBlockReadData()**: This function can be used to read a block of data bytes from the Receive FIFO.

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()** 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:

- The function **u16AHI_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)

Flow control can be implemented automatically in UART0 4-wire mode, 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.

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

To transmit data, the sending application can use one of the following functions:

- **vAHI_UartWriteData()**: This function can be used to write a single byte of data to the Transmit FIFO. If used, this function may be called multiple times to queue data bytes for transmission.
- **u16AHI_UartBlockReadData()**: This function can be used to read a block of data bytes from the Receive FIFO. The function will return the number of bytes that have been successfully written to the FIFO.

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. The transfer of data from the FIFO to the TxD line is handled automatically by the DMA engine.

Note that before calling **vAHI_UartWriteData()** or **u16AHI_UartBlockReadData()** 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()** or **u8AHI_UartReadLineStatus()**.

Receiving Data

Data is received on the RxD line on the destination device. The local transfer of data from the RxD line to the Receive FIFO is handled automatically by the DMA engine. The received data can be read from the Receive FIFO using one of the following functions:

- **u8AHI_UartReadData()**: This function can be used to read a single byte of data from the Receive FIFO.
- **u16AHI_UartBlockReadData()**: This function can be used to read a block of data bytes from the Receive FIFO.

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

- The function **u16AHI_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 Transmitting Serial Data in 1-wire Mode

In 1-wire mode, the UARTs use only the TxD line and can only transmit serial data. This transmission is performed at the convenience of the sending device (so no flow control is implemented).

For this mode, the function **vAHI_SetDIOPinMultiplexValue()** must be called to route the TXD pin of the UART to be used to the required DIO pin before the UART is enabled using **bAHI_UartEnableNoneDIO()**. Since the RxD line is not used and the Receive FIFO buffer is therefore not needed, in **bAHI_UartEnableNoneDIO()** you are advised to set the pointer to the start of this buffer in RAM to NULL - this avoids allocating RAM space to this buffer.

6.7 Break Condition

During a data transfer, 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 JN517x 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 6.8.

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.

6.8 UART Interrupt Handling

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

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

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

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

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

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

> **Note:** If the Application Queue API is being used, the above issue with the UART interrupts is handled by this API, so the application does not need to deal with it. For more information on this API, refer to the *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*.

7. Timers

This chapter describes control of the on-chip timers using functions of the Integrated Peripherals API. On the JN517x device, there are nine timers:

- Timer 0
- Timer 1
- PWM Timer 1 (also known as Timer 2)
- PWM Timer 2 (also known as Timer 3)
- PWM Timer 3 (also known as Timer 4)
- PWM Timer 4 (also known as Timer 5)
- PWM Timer 5 (also known as Timer 6)
- PWM Timer 6 (also known as Timer 7)
- Analogue Peripheral Timer (APT, also known as Timer 8)

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

The timers offer a range of operating modes:

- Timer mode
- Pulse Width Modulation (PWM) mode
- Counter mode
- Capture mode
- Delta-Sigma mode

However, not all the timers can operate in all modes. PWM Timers 1-6 and APT do not support modes that require external inputs - these are Counter mode and Capture mode. The timer modes are outlined in Section 7.1.

To use a timer in one of the above 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 following timer modes are available on the JN517x microcontrollers: 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_TimerEnableNoneDIO()** or **vAHI_TimerEnable()**). A mode is supported by all JN517x timers unless otherwise stated.

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. **Supported by Timers 0 & 1 and PWM Timers 1-6**	**vAHI_TimerConfigureOutputs()** **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. **Supported by Timers 0 & 1 and PWM Timers 1-6**	**vAHI_TimerConfigureOutputs()** **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. **Supported by Timer 0**	**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. **Supported by Timer 0 & 1**	**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. **Supported by Timers 0 & 1 and PWM Timers 1-6**	**vAHI_TimerStartDeltaSigma()**

Table 3: 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	Timer Number *	Timer Output	Capture Input	Clock Gate
Timer 0	0	TIM0_OUT DIO3/8	TIM0_CAP DIO2/5	TIM0_CKGT DIO4/15
Timer 1	1	TIM1_OUT DIO11/18	TIM1_CAP DIO6/17	Not Available
PWM Timer 1	2	PWM1 DIO12	Not Available	Not Available
PWM Timer 2	3	PWM2 DIO13	Not Available	Not Available
PWM Timer 3	4	PWM3 DIO14	Not Available	Not Available
PWM Timer 4	5	PWM4 DIO3	Not Available	Not Available
PWM Timer 5	6	PWM5 DIO8	Not Available	Not Available
PWM Timer 6	7	PWM6 DIO15	Not Available	Not Available
APT (Timer 8)	8	Not Available	Not Available	Not Available

Table 4: DIO Usage with JN517x Timers

* This value is used in the *u8Timer* parameter of the timer enable functions (see Section 7.2.2).

Timer inputs and outputs must be selected using the function **vAHI_SetDIOpinMultiplexValue()** before the timer is itself selected and enabled with **vAHI_TimerEnableNoneDIO()** or **vAHI_TimerEnable()**, described in Section 7.2.2.

Note: The function **vAHI_TimerEnable()** itself configures the DIO for the output of PWM Timers 1-3. When using this function for one of these PWM timers, there is no need to call the DIO selection function.

Caution: *The above DIO configuration should be performed before a timer is enabled, 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 selected, configured and enabled using the function **vAHI_TimerEnableNoneDIO()** or **vAHI_TimerEnable()**. A timer is selected by specifying the timer number through the function parameter *u8Timer* - the values of this parameter for the different timers are indicated in Table 4 on page 79.

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

The **vAHI_TimerEnableNoneDIO()** and **vAHI_TimerEnable()** functions contain certain configuration parameters, which are outlined below.

- **Clock Divisor:**

  To obtain the timer frequency, the peripheral 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 16MHz peripheral clock and a *prescale* value of 3, a division factor of 8 is used to give a timer frequency of 2MHz. A system clock sourced from the external crystal oscillator will give the most stable timer frequency (for system clock options, refer to Section 3.1).

- **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, DIO3 or 8 can be used
  - For Timer 1, DIO11 or 18 can be used
  - For PWM Timer 1, DIO12 is used
  - For PWM Timer 2, DIO13 is used
  - For PWM Timer 3, DIO14 is used
  - For PWM Timer 4, DIO3 is used
  - For PWM Timer 5, DIO8 is used
  - For PWM Timer 6, DIO15 is used

Once a timer has been enabled using **vAHI_TimerEnableNoneDIO()** or **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 Clocks

Each timer requires a source clock, which is provided by the peripheral clock. This source clock is divided down to produce the timer's clock. The division factor is specified when the timer is enabled - see Section 7.2.2. A system clock sourced from the external crystal oscillator gives the most stable timer frequency (for system clock options, refer to Section 3.1).

When Timer 0 is operating in Counter mode (see Section 7.3.4), an external clock is monitored by the timer. This signal can be fed into the TIM0CK_GT input on either of the DIO4 or DIO15 pins, selected using **vAHI_SetDIOpinMultiplexValue()**, and this input must be enabled using the function **vAHI_TimerClockSelect()** which must be called after **vAHI_TimerEnableNoneDIO()** or **vAHI_TimerEnable()**.

Normally PWM Timers are clocked from the 16MHz system clock. However, to provide higher resolution timing for these timers, it is possible to switch to using the 32 MHz clock using the function **bAHI_Switch32MHzClockForPWM()**. Since the timers are now clocking at twice the rate, the timers have been extended to 17 bits instead of the original 16-bits as found in previous families of JN51xx devices, and **bAHI_Switch32MHzClockForPWM()** enables the extra bit. As a consequence, a number of new read functions have been introduced to cope with this extension in range - these functions are **u32AHI_Timer17bitReadCount()**, **vAHI_Timer17bitReadCapture()** and **vAHI_Timer17bitReadCaptureFreeRunning()**. These functions must be used to read timers operating at 32 MHz in order to ensure that the correct value is read back. Note that Timer 5 (PWM4) does not support the use of a 32MHz clock.

In normal operating mode using the 16 MHz clock, the original versions of the above read functions (**u16AHI_TimerReadCount()**, **vAHI_TimerReadCapture()** and **vAHI_TimerReadCaptureFreeRunning()**) should be used.

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 (Timer 0 only) - see Section 7.3.3
- Counter mode (Timer 0 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.

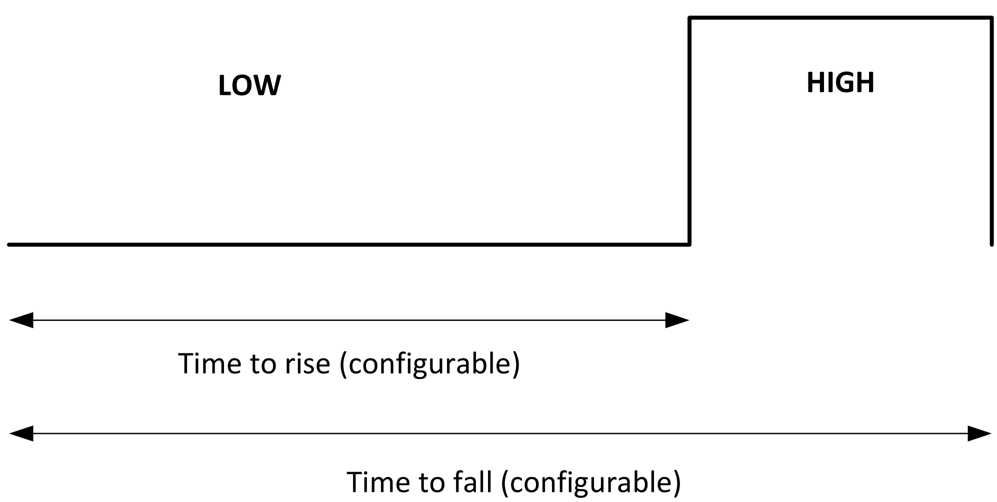


Figure 7: Timer Mode Signal

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 - see Section 7.2.1 for the relevant DIOs. This output can be enabled in **vAHI_TimerEnableNoneDIO()** or **vAHI_TimerEnable()**. The output can also be inverted using the function **vAHI_TimerConfigureOutputs()**.

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 on a DIO pin to be enabled in the call to **vAHI_TimerEnable()** or **vAHI_TimerEnableNoneDIO()** - see Section 7.2.1 for the DIOs. 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 16MHz peripheral 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) - 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 JN517x Data Sheet.

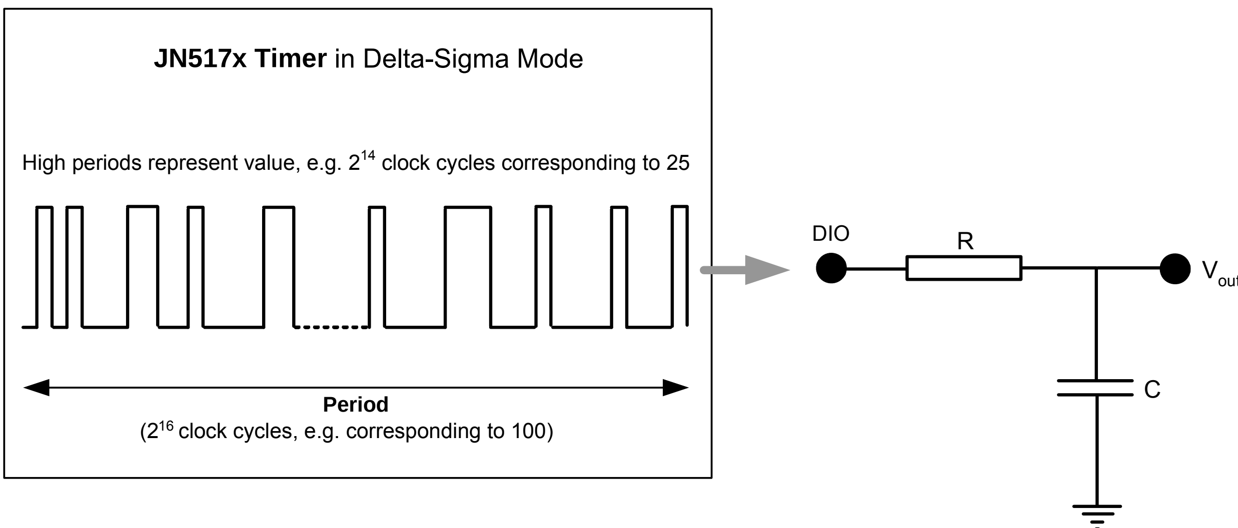


Figure 8: Delta-Sigma NRZ Mode Operation

7.3.3 Capture Mode

Capture mode is available on Timers 0 and 1 only. In this mode, the timer can be used to measure the pulse width of an external input. The external signal must be provided on a DIO pin (TIMx_CAP) - see Section 7.2.1 for the DIOs. 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_TimerEnableNoneDIO()** or **vAHI_TimerEnable()**.

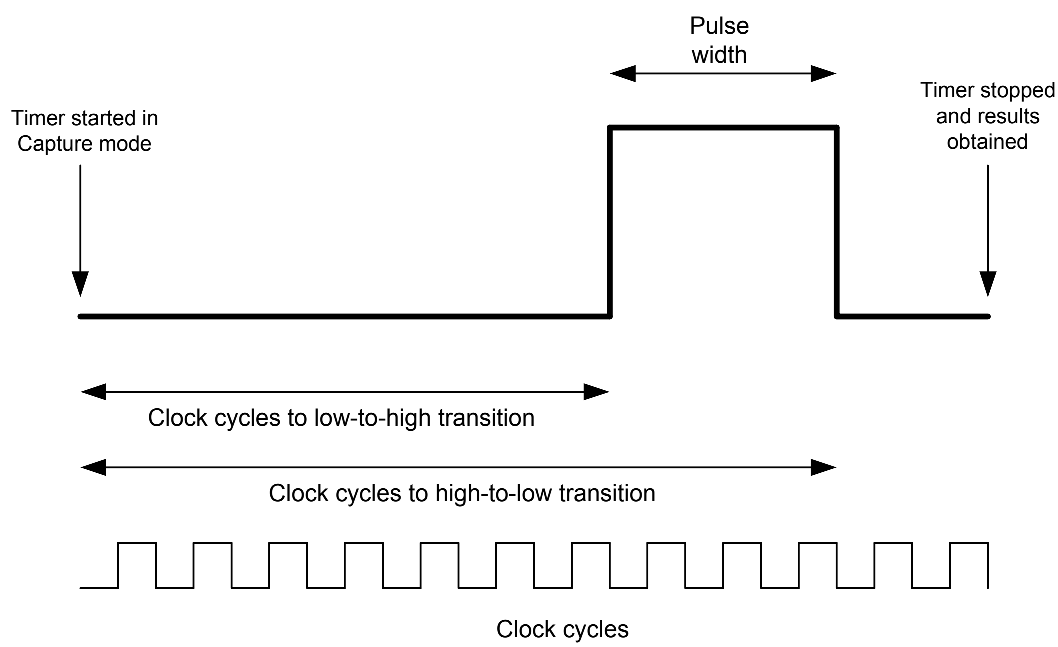


Figure 9: Capture Mode Operation

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

Counter mode is available on Timer 0 only. In this mode, the timer counts edges on an external clock signal, which must be provided on a DIO pin (TIM0_CKGT) - see Section 7.2.1 for the DIOs. 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 100ns apart, i.e. pulses must be wider than 100ns.

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_TimerEnableNoneDIO()** or **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_TimerEnableNoneDIO()** or **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 can be registered using dedicated registration functions for the individual timers:

- **vAHI_Timer0RegisterCallback()** for Timer 0
- **vAHI_Timer1RegisterCallback()** for Timer 1
- **vAHI_Timer2RegisterCallback()** for PWM Timer 1
- **vAHI_Timer3RegisterCallback()** for PWM Timer 2
- **vAHI_Timer4RegisterCallback()** for PWM Timer 3
- **vAHI_Timer5RegisterCallback()** for PWM Timer 4
- **vAHI_Timer6RegisterCallback()** for PWM Timer 5
- **vAHI_Timer7RegisterCallback()** for PWM Timer 6
- **vAHI_Timer8RegisterCallback()** for Analogue Peripheral Timer

Alternatively, the function **boAHI_RegisterTimerCallbackById()** can be used to register a callback function for any individual timer.

The relevant callback function is automatically invoked when an interrupt of the type E_AHI_DEVICE_TIMERx (where x is 0 to 8) occurs. The exact nature of the interrupt (from the two conditions listed above) can then be identified from a bitmap that is passed into the function. Note that the interrupt will be automatically cleared before the callback function is invoked.

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

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

8. Wake Timers

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

The JN517x microcontroller includes two wake timers, denoted Wake Timer 0 and Wake Timer 1, where each is a 41-bit counter. The wake timers are based on the 32kHz clock (which can be sourced internally or externally, as described in Section 3.1.5) 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.

A wake timer can then be started using the function **vAHI_WakeTimerStartLarge()**. This function takes as a parameter the starting value for the countdown - this value must be specified in 32kHz clock periods (thus, 32 corresponds to 1 millisecond).

On reaching zero, the timer 'fires', rolls over to 0x1FFFFFFFFFF and continues to count down. If enabled, the wake timer interrupt is generated on reaching zero.

> **Note:** If the 32kHz clock is sourced from the (default) internal 32kHz RC oscillator, the wake timers may run up to 18% fast or slow. For more 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 the function **u64AHI_WakeTimerReadLarge()**. This function does 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 JN517x microcontroller's 32kHz clock. If this clock is sourced from the internal 32kHz RC oscillator, it may run up to 40% fast or 10% slow on the JN517x device, depending on temperature, supply voltage and manufacturing tolerance. To achieve more accurate timings in this case, the self-calibration facility should be used that compares the 32kHz clock against the faster and more accurate peripheral clock, which should be running at 16MHz with the system clock sourced from the external crystal oscillator (for system clock information, refer to Section 3.1). This test is performed using Wake Timer 0. The result of this calibration allows you to calculate the required number of 32kHz clock cycles to achieve the desired timer duration when starting a wake timer with the function **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 32kHz clock. At the same time, a reference counter starts counting up from zero using the 16MHz peripheral clock.

4. When the wake timer reaches zero, **u32AHI_WakeTimerCalibrate()** returns the number of 16MHz clock cycles registered by the reference counter. Let this value be *n*.

- If the clock is running at 32kHz, $n = 10000$
- If the clock is running slower than 32kHz, $n > 10000$
- If the clock is running faster than 32kHz, $n < 10000$

5. You can then calculate the required number of 32kHz clock periods (for **vAHI_WakeTimerStartLarge()**) to achieve the desired timer duration. If *T* is the required duration in seconds, the appropriate number of 32kHz 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 32kHz 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 32kHz clock periods than to over-estimate them.

Note: For the above calibration, you can alternatively use the **u32AHI_WakeTimerCalibrateEnhanced()** function, which allows you to specify the number of 32kHz clock periods over which the calibration will be run. Refer to the function description for details.

9. Tick Timer

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

The Tick Timer is a 24-bit hardware timer which can clocked from the System Clock, derived from the CPU clock, which in turn is driven from either the high-speed RC Oscillator or a 32MHz crystal. Alternatively, it may be driven from an external 32kHz clock. The Tick Timer can be used to implement:

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

Note: For high-precision Tick Timer operation, the peripheral clock should run at 16MHz with the system clock sourced from the external crystal oscillator. For system clock information, refer to Section 3.1.

9.1 Tick Timer Operation

The Tick Timer counts downwards from a pre-programmed reference value until the count matches zero. The timer may operate in one of two modes, which determine what the timer will do once zero has been reached. The options are:

- Restart the count from the reference value
- Stop counting on reaching 0 (single-shot mode)

An interrupt can also be enabled which is generated on reaching zero.

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 functions **vAHI_TickTimerStop()** to ensure the timer is stopped, and then **vAHI_SetTickTimerSource()** to select the clock to be used.

The operational mode of the Tick Timer is specified by calling the function **vAHI_TickTimerConfigure()**. The operational mode can be to disable the Tick Timer, stop the Tick Timer or restart the timer from the previously programmed timer interval.

Next the timer interval is set using the function **vAHI_TickTimerInterval()**, specifying the number of clocks in the required interval (in the range 0 to 0xFFFFFF).

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 v**AHI_TickTimerInit()** to register an interrupt callback function and enable the timer.

Note that if the Tick Timer is started in single-shot mode, once it has stopped (on reaching zero) it can be started again simply by setting another starting value using **vAHI_TickTimerInterval()**.

The timer can be stopped by calling the function **vAHI_TickTimerStop()**.

The current value of the Tick Timer can be read by calling **u32AHI_TickTimerRead()** and can be cleared by calling **vAHI_TickTimerClear()**.

9.3 Tick Timer Interrupts

An interrupt can be enabled that will be generated when the Tick Timer reaches zero. This interrupt is enabled or disabled using the function **vAHI_TickTimerIntEnable()**.

The Tick Timer interrupt can invoke a user-defined callback function which is registered using the function **vAHI_TickTimerInit()**.

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

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

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

- **bAHI_TickTimerIntStatus()** obtains the current interrupt status of the Tick Timer.
- **vAHI_TickTimerIntPendClr()** clears a pending Tick Timer interrupt.
- **bAHI_TickTimerWrapStatus()** indicates when the Tick Timer has counted from 1 to 0, i.e. made the transition to zero.

10. Watchdog Timer

This chapter describes control of the Watchdog Timer on the JN517x device using functions of the Integrated Peripherals API.

The Watchdog Timer is provided to allow the JN517x device to recover from software lock-ups. Note that a watchdog can also be implemented using the Tick Timer, described in Chapter 9.

10.1 Watchdog Operation

The Watchdog Timer implements a timeout period and is derived from the internal high-speed RC oscillator (which runs at 27MHz or 32MHz).

On reaching the timeout period, the JN517x 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 JN517x 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 1: Following a power-up, reset or wake-up from sleep, the Watchdog Timer is enabled with the maximum possible timeout period of 16392ms (regardless of its state before any sleep or reset).

Note 2: The Watchdog Timer can be configured to invoke an exception on timeout. This allows debugging of the situation that led to the timeout during application development. For more information, refer to Section 10.2.3.

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 JN517x device. It is started with the maximum possible timeout of 16392ms.

- 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 = 8ms   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 16392ms.

Note that if the Watchdog Timer is sourced from an internal RC oscillator, the actual timeout period obtained may be up to 18% less than the calculated value due to variations in the 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.

Caution: *Be sure to set the Watchdog timeout period to be greater than the worst-case Flash memory read-write cycle. If the Watchdog times out during a Flash memory access, the JN517x microcontroller will enter programming mode. For information on read-write cycle times, refer to the relevant Flash memory data sheet.*

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 18% shorter than the calculated timeout period (if the timer is sourced from an internal RC oscillator - 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()**.

10.2.3 Exception Handler for Debug

By default, the expiry of the Watchdog Timer will cause a reset of the JN517x device. Alternatively, an exception can be invoked on expiry of the timer. The exception is serviced by the stack overflow exception handler, which can call the function **bAHI_WatchdogResetEvent()** to determine if a Watchdog exception occurred. This may help to debug the situation which led to the Watchdog timeout. Therefore, this option is designed for use only during application development.

The exception option is enabled by calling the function **vAHI_WatchdogException()**. The stack overflow exception handler function should be developed before enabling the Watchdog exception option.

Note: The stack overflow exception handler function should have the following prototype definition:

PUBLIC void vException_StackOverflow(void);

We would not expect an exception handler written in C to return - once it has performed any actions, it should either sit in a loop or reset the device.

11. Pulse Counters

This chapter describes control of the pulse counters on the JN517x device using functions of the Integrated Peripherals API.

Two pulse counters are provided on the JN517x 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, Pulse Counter 0 and Pulse Counter 1, are each 16-bit counters. On the JN517x device, Pulse Counter 0 may receive its input signal on pins DIO1 or DIO13 and Pulse Counter 1 may receive its input on DIO5 or DIO14. The pin to be used as the input from the two available for each Pulse Counter is selected by calling **vAHI_SetDIOpinMultiplexValue()** before the Pulse Counter is enabled.

The two counters can be combined together to form a single 32-bit counter, if desired - in this case, the DIO on which the input signal is taken can be selected from the input pins for the two counters.

The pulse counters can operate in all power modes of the JN517x device, including sleep, and with input signals of up to 100kHz. An increment of the counter can be configured to occur on a rising or falling edge of the relevant input. Each pulse counter has an associated user-defined reference value. An interrupt (or wake-up event, if asleep) can be generated when the counter passes its pre-configured reference value - that is, when the count reaches *(reference value + 1)*. The counters do not saturate at their maximum count values, but wrap around to zero.

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

Debounce

The input pulses can be debounced using the 32kHz 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:

- 100kHz, if debounce disabled
- 3.7kHz, if debounce enabled to operate with 2 consecutive samples
- 2.2kHz, if debounce enabled to operate with 4 consecutive samples
- 1.2kHz, 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 32kHz 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

Before a pulse counter is configured, the input signal pin must be selected using the function **vAHI_SetDIOpinMultiplexValue()**. The pulse counter can then 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 and, if so, the pin on which the combined counter may take its input from either of the selected inputs. It is also possible to route only one of the two input pins to the combined counter using **vAHI_SetDIOpinMultiplexValue()**
- if the pulse count is to be incremented on the rising edge or falling edge of a pulse in the input signal
- if the debounce feature is to be enabled and, if so, the number of consecutive samples (2, 4 or 8) with which it will operate (see Section 11.1)
- if an interrupt is to be enabled which is generated when the pulse count passes the reference value (see below)

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

11.2.2 Starting and Stopping a Pulse Counter

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

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

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

11.2.3 Monitoring a Pulse Counter

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

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

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

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

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

11.3 Pulse Counter Interrupts

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

Note: A pulse counter continues to run during sleep. A pulse counter interrupt can be used to wake the JN517x 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. Infra-Red Transmitter

This chapter describes control of the infra-red transmitter on the JN517x device using functions of the Integrated Peripherals API.

Infra-red transmission is a special feature of PWM Timer 4 (also known as Timer 5) in which the timer is used to generate waveforms for infra-red remote control applications.

12.1 Infra-Red Transmitter Operation

Consumer equipment remote control protocols, such as Philips RC-6, apply On-Off Key (OOK) modulation to a carrier signal using an encoded bit stream. The infra-red transmitter is able to accommodate a variety of remote control protocols that have different carrier frequency, carrier duty-cycle and data bit encoding requirements. The infra-red transmitter uses PWM Timer 4 to produce a programmable carrier waveform that is OOK modulated by a programmable bit sequence stored in RAM. The resultant waveform is output to the associated PWM Timer 4 (IR) signal, multiplexed on the DIO3 output pin.

> ***Caution:*** *A typical infra-red LED requires at least 15mA of drive current. An external transistor or LED driver will be required to supply this current because the standard digital outputs do not have sufficient drive strength.*

Example Waveform

An example of an OOK modulated waveform is shown in Figure 10.

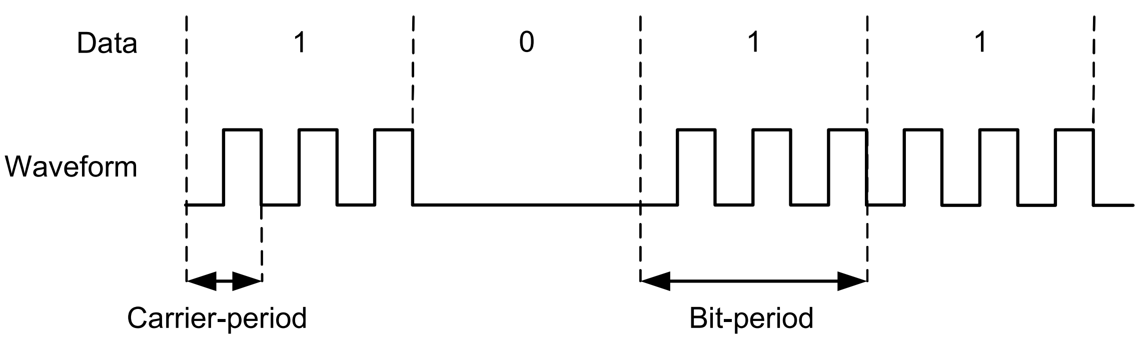


Figure 10: Example OOK Modulated Waveform

In this example, the resultant OOK modulated waveform is generated from the logical AND of a periodic carrier signal and the binary bit pattern 1011, where the period of each data bit is exactly equal to three times the carrier-period.

12.2 Using the Infra-Red Transmitter

This section describes how to use the Integrated Peripherals API functions to configure, start and monitor an infra-red transmission.

12.2.1 Configuring the Infra-Red Transmitter

The infra-red transmitter must first be enabled and configured using the **bAHI_InfraredEnable()** function. This function call must specify:

- the clock prescale value used to divide down the peripheral clock and produce the timer clock
- the number of timer clock periods after starting the timer before the carrier goes high - this defines the carrier low duration
- the number of timer clock periods after starting the timer before the carrier goes low again - this defines the carrier period
- the bit-period in units of the carrier period
- the output signal polarity
- if an interrupt is to be enabled that indicates the end of transmission

Example Configuration

The Philips RC-6 protocol requires a carrier signal of 36kHz ±10% with a duty cycle between 25% and 50%. In this example we will use a duty cycle of 30%.

The RC-6 protocol encodes logic ‘0’ as bits ‘01’, logic ‘1’ as bits ‘10’ and the leader symbol as bits ‘11111100’. Each individual bit has a duration of 16 times the carrier period (i.e. 16x1/36kHz ≈ 444µs). These waveform timing requirements can be satisfied by calling **bAHI_InfraredEnable()** with the following input parameter values:

- *u8Prescale*: 2 (timer clock period = $2^{u8Prescale}$/16MHz = 4/16MHz = 250ns)
- *u16Hi*: 78 (carrier low duration = 78x250ns = 19.5µs)
- *u16Lo*: 111 (carrier period = 111x250ns = 27.75µs, i.e. frequency = 36.036kHz)
- *u16BitPeriodInCarrierPeriods*: 16 (bit period = 16x27.75µs = 444µs)
- *bInvertOutput*: TRUE or FALSE as required by the external transistor
- *bInterruptEnable*: TRUE or FALSE as required by the application

Note: To guarantee accurate waveform timings the user is advised to ensure that the peripheral clock operates at 16MHz with the system clock sourced from the external crystal oscillator - see Section 3.1.

12.2.2 Starting an Infra-Red Transmission

Having configured the waveform timing requirements for the remote control protocol by calling **bAHI_InfraredEnable()**, the user can start the waveform generation for the infra-red transmission by calling the function **bAHI_InfraredStart()**. This function call must specify:

- the start address in RAM of a 32-bit wide array containing the encoded bits to be transmitted (maximum array size of 128 words)
- the number of encoded bits from the array to be transmitted (1 to 4096 bits)

Prior to calling **bAHI_InfraredStart()**, the user should populate the data array with the required encoded bit pattern to be transmitted. The MSB of each 32-bit word will be transmitted first. For example, a transmission of 35 bits will require the user to program all 32 bits in the first 32-bit word followed by the upper 3 bits in the second 32-bit word.

Note: The data array should contain an encoded bit sequence. It is the responsibility of the application to perform this encoding as required by the protocol.

On calling **bAHI_InfraredStart()**, waveform generation will start - the device will automatically read the specified number of bits from the data array using a DMA mechanism and produce an OOK modulated carrier waveform using the pre-configured timing requirements.

The generated waveform will be available on the PWM4 output which can be directed to pin DIO3 using the function **vAHI_SetDIOpinMultiplexValue()**.

Note: To prevent glitches from occurring on the output pins associated with PWM Timer 4, we recommend that the application calls the function **vAHI_SetDIOpinMultiplexValue()** before **bAHI_InfraredEnable()**.

12.2.3 Monitoring an Infra-Red Transmission

The application can detect when an infra-red transmission has completed in either of the following ways:

- An interrupt can be enabled which is triggered when the transmission completes (see Section 12.3)
- The application can use the function **bAHI_InfraredStatus()** to poll the infra-red transmission status - this function returns TRUE if a transmission is in progress and FALSE otherwise

12.2.4 Disabling the Infra-Red Transmitter

If enabled, the infra-red transmitter may be subsequently disabled by calling the function **vAHI_InfraredDisable()**. After calling this function, **bAHI_InfraredEnable()** must first be called before attempting to call any other infra-red function.

12.3 Infra-Red Transmitter Interrupt

The infra-red transmitter can optionally generate an interrupt when the infra-red transmission has completed. This interrupt can be enabled as part of the call to the function **bAHI_InfraredEnable()**.

The interrupt is handled as an Infra-Red Transmitter interrupt and must be incorporated in the user-defined callback function registered using the function **vAHI_InfraredRegisterCallback()**.

The registered callback function is automatically invoked when an interrupt of the type E_AHI_DEVICE_INFRARED occurs. The source of the interrupt will be indicated in the bitmap that is passed into the callback function. Note that the interrupt will be automatically cleared before the callback function is invoked.

13. Inter-Integrated Circuit (I^{2}C) Interface

This chapter describes control of the 2-wire I^{2}C Interface (also known as Serial Interface or SI) using functions of the Integrated Peripherals API.

The JN517x microcontroller includes an industry-standard I^{2}C synchronous serial interface that provides a simple and efficient method of data exchange between devices. The I^{2}C Interface comprises two signals:

- Serial Data (SDA) line
- Serial Clock (SCL) line

The I^{2}C peripheral on a JN517x device can act as a master or a slave of the I^{2}C bus:

- I^{2}C master functionality is described in Section 13.2
- I^{2}C slave functionality is described in Section 13.3

I^{2}C configuration which is common to the master and slave functionality is described in Section 13.1.

> **Tip:** The protocol used by the interface is detailed in the I^{2}C Specification (available from www.nxp.com).

13.1 Common Configuration

There are several functions which are common to both the master and slave functionality, based around the I/O lines and interrupts.

The I^{2}C interface signals are available on either of the following pairs of pins:

- DIO2 (SDA) and DIO3 (SCL)
- DIO5 (SDA) and DIO4 (SCL)

The first I^{2}C Interface function to be called is **vAHI_I2CcontrollerConfigure()** which allows the DIOs for the SDA and SCL signals to be selected and configured for I^{2}C use. The function enables the open-drain feature on the DIO4 and DIO5 pins, but not on DIO2 and DIO3. It also allows slew control and spike suppression to be enabled or disabled on each of the selected pins (pulse suppression filters on the clock and data lines suppress any spurious high or low pulses with a pulse width less than 60ns).

If the device is configured to use the I^{2}C interface over DIO4 and DIO5 but is then reconfigured such that DIO4 and DIO5 are reassigned to other uses, the function **vAHI_SetI2CpinsToMFIOmode()** should be called to ensure that the open-drain, slew rate control and spike suppression functionalities are no longer active on these pins. Failure to do so may impair the ability of the device to drive correct logic values on the pins.

Settings for the I/O cell (open drain or MFIO) and slew rate control are retained over sleep modes which allow a warm restart of the device. The spike suppression setting will not be held over any sleep period and will default to disabled when the chip wakes.

If spike suppression is required, it should be re-enabled after sleep by calling **vAHI_I2CcontrollerConfigure()**.

Sleep periods in which the chip is goes into a deep sleep mode, which requires the device to perform a cold start at restart, will not be affected. The cold start will perform all the initialisation routines for the device which will include the I^{2}C configuration if this functionality is in use

13.2 I^{2}C Master

The I^{2}C master can implement communication in either direction with a slave device on the I^{2}C bus. This section describes how to implement a data transfer.

> **Note:** The I^{2}C bus on the JN517x 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.

13.2.1 Enabling the I^2C Master

The I^2C master has its own set of functions in the Integrated Peripherals API (and the I^2C slave has a separate set of functions). Before using any of the I^2C master functions, the I^2C master peripheral must be enabled using the function **vAHI_I2CMasterConfigure()**.

When enabled, this interface uses the pins selected for the clock line and bi-directional data line specified in the call to **vAHI_SetDIOpinMultiplexValue()** as described in Section 5.1.6.

As a bus master, the microcontroller provides the clock (on the clock line) for synchronous data transfers (on the data line), where the clock frequency is set as part of the configuration specified by **vAHI_I2CMasterConfigure()**. The I^2C clock is derived from the system peripheral clock which must run at 16MHz (the system clock must be sourced from an external crystal oscillator - for system clock information, refer to Section 3.1).

The I^2C enable functions also allow I^2C interrupts (of the type E_AHI_DEVICE_I2C) to be enabled, which are handled by the user-defined callback function registered using the function **vAHI_I2CRegisterCallback()**. For details of the callback function prototype, refer to Appendix A.1.

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

13.2.2 Writing Data to a Slave

The I^2C master can write data to an I^2C slave which has a 7-bit or 10-bit address. Slave transfers (both writing to the slave and reading from a slave) use the function **u16AHI_I2CMasterReadWriteData()**. To write information to a slave, the function is called with parameters which include:

- The slave address and an indication whether it is 7 bits or 10 bits in length
- A pointer to a buffer containing the data to be written
- The number of bytes to be written
- The direction of the transfer (write)

The function returns the number of number of bytes transferred.

13.2.3 Reading Data from a Slave

The I^2C master can read data sent from an I^2C slave which has a 7-bit or 10-bit address. Slave transfers (both writing to the slave and reading from a slave) use the function **u16AHI_I2CMasterReadWriteData()**. To read information from a slave, the function is called with parameters which include:

- The slave address and an indication whether it is a 7-bit or 10-bit address,
- A pointer to a buffer to hold the data received from the Slave
- The direction of the transfer (read)

The number of bytes transferred in the transaction is returned.

13.2.4 Multiple (Block Chain) Transfers

It is possible to perform a number of transfers one after the other, with the master keeping control of the I^2C bus, by using the function **iAHI_I2CMasterMultiReadWriteData()**. This function takes as a parameter a pointer to a linked list of Block Chain Transfers (BCT), of type **tsAHII2CBCTentry**, which must be set up before the function is called. A BCT entry contains the same information provided to the **u16AHI_I2CMasterReadWriteData()** above, allowing multiple transfers to be specified by setting up a linked list with the information for each individual transfer. The other parameter required is the number of transfers held in the list. The function returns the number of bytes sent for this invocation or an error code. If it is not possible to complete all the transfers because the internal transfer buffer of the master has been filled, the function will return with the number of bytes it was able to put in the buffer. The remainder of the transfer will be completed under interrupt control - the result of the transfer can be a value provided by the interrupt callback. It is possible to check whether the whole Block Chain was transferred by calling the function **boAHI_I2CIsMasterTransferComplete()**, which returns TRUE if all the transfers were completed.

13.2.5 Waiting for Completion

Slave write and read operations (see Section 13.2.2, Section 13.2.3 and Section 13.2.4) may not complete in the time that the initiating function **boAHI_I2CIsMasterTransferComplete()** or **iAHI_I2CMasterMultiReadWriteData()** executes. In this case, if further calls to this function are to be made, the function will fail if a previous transfer is still in progress.

The user can continue to call the function until it is accepted or, alternatively, the function **boAHI_I2CIsMasterTransferComplete()** can be used to check the state of the transfer.

13.2.6 Interrupts

Interrupts are enabled on the I^2C interface using the function **vAHI_I2CMasterConfigure()**. I^2C interrupts (of the type E_AHI_DEVICE_I2C) are handled by the user-defined callback function registered using the function **vAHI_I2CRegisterCallback()**. For details of the callback function prototype, refer to Appendix A.1.

The callback receives information from the interrupt handler on the condition(s) which caused the interrupt. For the I^2C master, the following conditions are available:

- **User Rx Buffer Full:** No further writes can occur
- **Receive FIFO Data Available Interrupt (RFDA):** RX_FIFO has received data
- **Receive FIFO Full Interrupt (RFF):** RX_FIFO is full. When more I^2C data is received, the SCL signal is stretched
- **Master Transmitter Data Request Interrupt (MTDR):** Transmission has started and TX_FIFO needs data. When TX_FIFO is empty, the SCL signal is stretched
- **I^2C error (IBE):** An error has occurred on the I^2C bus
- **Master Transmitter No Acknowledge Interrupt (MTNA):** The master transmitter has received a 'no acknowledge' from the addressed slave
- **Transmitter Arbitration Failure Interrupt (TAF):** The transmitter (master or slave) has lost arbitration
- **Master Transaction Done Interrupt (MTD):** The master has completed a transaction successfully

In many cases, these conditions are dealt with transparently by the interrupt handler (e.g. MTDR is used to cause the interrupt handler to fill up the transmit FIFO with more data from the user transmit buffer, and RFF is used to trigger the unloading of the RX_FIFO into the user receive buffer).

13.3 I^2C Slave

The I^2C peripheral on the JN517x device can act as an I^2C master or an I^2C slave (but not as both at the same time). This section describes what must be done to allow the I^2C slave to participate in a data transfer initiated by a remote I^2C master.

13.3.1 Enabling the Slave and its Interrupts

The I^2C slave must first be configured and enabled using the function **vAHI_I2CSlaveConfigure()**. This function requires the address size of the slave to be specified as 7-bit or 10-bit, and the I^2C slave address itself to be specified. The function also allows the generation of I^2C slave interrupts to be configured.

I^2C interrupts (of the type E_AHI_DEVICE_I2C) are handled by the user-defined callback function registered using the function **vAHI_I2CRegisterCallback()**. This is the same registration function as used for the I^2C master. For details of the callback function prototype, refer to Appendix A.1.

The callback receives information from the interrupt handler on the condition(s) which caused the interrupt. For the I^2C slave, the following conditions are available:

- **User Rx Buffer Full:** No further data can be received
- **Receive FIFO Data Available (RFDA):** The hardware RX_FIFO has received data.
- **Receive FIFO Full Interrupt (RFF):** The hardware RX_FIFO is full. When more I2C data is received, the SCL signal is stretched.
- **Slave Transmitter Data Request Interrupt (STDR):** Transmission has started and the TXS_FIFO has run out of data. When the TXS_FIFO is empty, the SCL signal is stretched.
- **I2C error (IBE):** An error has occurred on the I2C-bus.
- **Slave RX Stop Detect Interrupt (SRSD):** The Slave receiver has detected a stop or restart condition.
- **Slave TX Stop Detect Interrupt (STSD):** The Slave transmitter has detected a stop or restart condition.
- **Transmitter Arbitration Failure Interrupt (TAF):** The transmitter (master or slave) has lost arbitration.

In many cases, these conditions are dealt with transparently by the interrupt handler (e.g. STDR is used to cause the interrupt handler to fill up the transmit FIFO with more data from the user transmit buffer, and RFF is used to trigger the unloading of the RX_FIFO into the user receive buffer).

> ***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.*

13.3.2 Receiving Data from the Master

An I^2C master indicates that it needs to send data to a particular I^2C slave as described in Section 13.2.2. The slave automatically responds to the 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 I^2C bus consists of a sequence of data bytes, beginning with a start of transmission and followed by the address of the slave for which the transfer is intended. When the address matches the slave address set up in **vAHI_I2CSlaveConfigure()**, the byte stream is captured and stored in the Slave Rx FIFO. When this FIFO is full or the end of transaction (stop) is detected, the contents are written into the user receive buffer set up by the call to **vAHI_I2CSlaveConfigure()**, under interrupt control. The user application is notified that a message is waiting in the buffer, or that the buffer is full.

13.3.3 Sending Data to the Master

An I^2C master indicates that it needs to obtain data from a particular I^2C slave as described in Section 13.2.3. The slave automatically responds to the 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 I^2C bus from slave to master is handled by the function **u16AHI_I2CSlaveWriteData()** which takes as parameters a pointer to a buffer containing the data to be transferred and the length (in bytes) of the message. The function returns the number of bytes of data that were initially sent to the Slave Tx FIFO. In the case of transfers which contain fewer bytes than the size of the Tx FIFO, the function returns the number of bytes transferred - in the case, where the amount of data to be transferred is larger than the Tx FIFO, the amount sent to fill up the Tx FIFO is returned. Subsequently, the I^2C interrupt handler will continue to fill up the Tx FIFO from the data buffer until either the transfer is complete or the Tx FIFO is again full. The end of the transfer is indicated by the interrupt callback routine showing that the SRSD (Slave Receiver Stop Detected) interrupt is active.

14. Serial Peripheral Interface (SPI) Master

This chapter describes control of the Serial Peripheral Interface (SPI) Master on the JN517x microcontroller using functions of the Integrated Peripherals API.

The Serial Peripheral Interface on the JN517x microcontroller allows high-speed synchronous data transfers between the microcontroller and peripheral devices, without software intervention. When the microcontroller operates as the master on the SPI bus, all other devices connected to the bus are expected to be slave devices under the control of the master's CPU.

14.1 SPI Bus Lines

The SPI Master provides the following signals: Clock (SPI_M_SCK), Data In (SPI_M_MISO) and Data Out (SPI_M_MOSI) - these signals are shared on the SPI bus.

Up to two slave-select output lines can be used simultaneously from the three available: SPI_M_SEL0, and one of SPI_M_SEL1 or SPI_M_SEL2, due to the need for SPI_M_MOSI to be present.

These signals are available on the pins shown in Table 5:

Signal	Pins
SPI_M_SCK	DO0, DIO11
SPI_M_MOSI	DIO7, DIO15
SPI_M_MISO	DO1, DIO18
SPI_M_SEL0	DIO6, DIO17
SPI_M_SEL1	DIO15
SPI_M_SEL2	DIO7

Table 5: DIO Usage with JN517x SPI Master

The location of the individual pins can be selected using a call to **vAHI_SetDIOpinMultiplexValue()** before enabling the SPI Master function with **vAHI_SpiConfigure()**.

14.2 Data Transfers

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 (shift register) in the device. Any data transaction size between 1 and 32 bits (inclusive) 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 peripheral clock (for system clock options, see Section 3.1), which may be divided down and allows bit rates of up to 16Mbps.

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

14.3 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 6: SPI Modes of Operation

14.4 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.

14.5 Using the Serial Peripheral Interface

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

14.5.1 Performing a Data Transfer

A SPI data transfer is performed as follows:

1. The SPI master must first be configured and enabled using the function **vAHI_SpiConfigure()**. This function allows the configuration of:
   - Number of SPI slaves
   - Clock divisor (for peripheral 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 14.6.
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()**. A transaction size between 1 and 32 bits can be specified.
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()**. The read data is aligned to the right (lower bits) of the returned 32-bit value.
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()**.

14.5.2 Performing a Continuous Transfer

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 14.5.1 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 returned 32-bit 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)**.

14.6 SPI Interrupts

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

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

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

15. Serial Peripheral Interface (SPI) Slave

This chapter describes control of the Serial Peripheral Interface (SPI) Slave on the JN517x microcontroller using functions of the Integrated Peripherals API.

The Serial Peripheral Interface on the JN517x microcontroller allows high-speed synchronous data transfers between the microcontroller and peripheral devices, without software intervention.

Note: The SPI Master device on the JN517x microcontroller is described in Chapter 14.

15.1 SPI Slave Operation

The SPI Slave is used for high-speed data exchanges between the JN517x microcontroller and a 'remote' processor, which may be a separate processor contained in the wireless network node. The remote processor must contain a SPI Master device, which initiates the data transfers. The data exchanges then require minimal CPU usage. Data transfer is full-duplex, so data is simultaneously transmitted and received by both communicating devices.

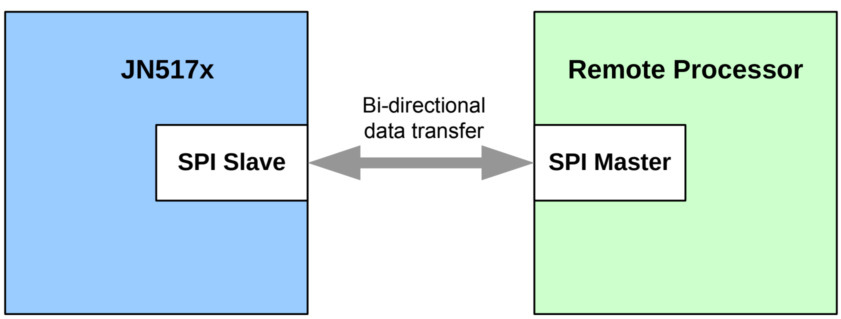


Figure 11: JN517x SPI Slave

The SPI Slave uses separate configurable FIFO buffers located in system RAM to store data bytes for transmission and reception.

Caution: *Only SPI mode 0 is supported. At both ends of the data link, the data to be transmitted is changed on a negative clock edge and received data is sampled on a positive clock edge.*

15.1.1 SPI Bus Lines and DIO Usage

The SPI Slave uses the following bus lines:

- Slave Clock Input, SPI_S_SCK
- Slave Data Output, SPI_S_MISO
- Slave Data Input, SPI_S_MOSI
- Slave-select Input, SPI_S_SEL

These signals may use the DIO pins shown in Table 7:

Signal	Pins
SPI_S_SCK	DIO6, DIO18
SPI_S_MOSI	DIO11, DIO14
SPI_S_MISO	DO1, DIO17
SPI_S_SEL	DIO7, DIO15

Table 7: DIO Usage with JN517x SPI Slave

The signals may be assigned to the DIO pins to be used with the function **vAHI_SetDIOpinMultiplexValue()** before enabling the SPI Slave by calling **bAHI_SpiSlaveEnable()**.

15.1.2 SPI Slave FIFOs and Interrupts

The Data In (Receive) and Data Out (Transmit) paths of the SPI Slave device contain FIFO buffers which are located in RAM. The exact locations and sizes of these buffers are defined by the application when the SPI Slave is initialised using the function **bAHI_SpiSlaveEnable()**. Each buffer can be up to 255 bytes in size.

Fill-level thresholds (in bytes) must also be specified that are used to prompt the application to write data to the Transmit buffer and read data from the Receive buffer.

- For the Transmit FIFO, this threshold is the fill-level which is considered low enough for more data to be written into the buffer - if interrupts are enabled, an interrupt will be generated when the amount of data in the buffer falls below this level
- For the Receive FIFO, this threshold is the fill-level which is considered high enough for data to be read from the buffer - if interrupts are enabled, an interrupt will be generated when the amount of data in the buffer rises above this level

A Receive timeout duration (in microseconds) must also be specified in the above function call. Following the end of a SPI transfer, if the Receive FIFO remains not empty for this duration then a timeout interrupt will be generated (if enabled) to prompt the application to read data from the buffer. This prevents received data from remaining in the buffer for too long without being read.

SPI Slave interrupts must be enabled in order to use the buffer thresholds and Receive timeout described above. Again, interrupts can be enabled when the device is configured using **bAHI_SpiSlaveEnable()**. If they are enabled, a user-defined callback function to handle SPI Slave interrupts must be registered using the function **vAHI_SpiSlaveRegisterCallback()**. The callback function is automatically invoked when an interrupt of the type E_AHI_DEVICE_SPIS occurs. For details of the callback function prototype, refer to Appendix A.1.

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

15.2 Using the SPI Slave

A data transfer is conducted via the SPI Slave as follows (this procedure assumes that SPI Slave interrupts will be enabled):

1. The SPI Slave must first be initialised and configured using the function **bAHI_SpiSlaveEnable()**. This function allows the following to be configured:
   - Bit-order for transmission/reception of SPI data (LSB first or MSB first)
   - DIO pins used for SPI_S_MISO and SPI_S_MOSI
   - Transmit FIFO buffer, including start address in RAM, size (in bytes) and write threshold (in bytes) - see Section 15.1.2
   - Receive FIFO buffer, including start address in RAM, size (in bytes), read threshold (in bytes) and timeout (in microseconds) - see Section 15.1.2
   - SPI Slave interrupts (which should be enabled)

2. A user-defined callback function to handle SPI Slave interrupts must now be registered using **vAHI_SpiSlaveRegisterCallback()**.

3. The application can now load transmission data into the Transmit FIFO (data will be transmitted when a transfer is initiated by the remote SPI Master):

   a) The initial data must be written to the Transmit FIFO using the function **vAHI_SpiSlaveTxWriteByte()**. The number of bytes written must not exceed the size of the buffer. By default, if the Transmit FIFO is empty and a transfer is initiated by the remote SPI Master, the SPI Slave will transmit the data byte 0x00.

   b) Subsequently, the application must wait for a write threshold interrupt to prompt further writes to the Transmit FIFO. When this interrupt occurs, the user-defined callback function will be invoked to handle the interrupt and **vAHI_SpiSlaveTxWriteByte()** should be called within this callback function. The number of bytes written should not exceed the size of the buffer minus the write threshold for the buffer.

4. The application can now also read any received data from the Receive FIFO. To do this, it should wait for a read threshold interrupt or a read timeout interrupt. When one of these interrupts occurs, the user-defined callback function will be invoked to handle the interrupt and the function **u8AHI_SpiSlaveRxReadByte()** should be called within this callback function.

The functions **u8AHI_SpiSlaveTxFillLevel()**, **u8AHI_SpiSlaveRxFillLevel()** and **u8AHI_SpiSlaveStatus()** are provided to enable an application to monitor the SPI Slave in a non-interrupt driven manner.

Note: The JN517x device requires that the SPI_S_SEL line is de-asserted by the SPI Master for at least 125 ns between consecutive byte transfers.

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

16. Flash Memory

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

The JN517x microcontroller has on-chip Flash memory. This non-volatile memory is used to store the binary application and associated application data. The JN517x device can also be optionally connected to an external Flash memory device.

The Integrated Peripherals API includes functions that allow the application to erase, programme and read a sector of 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.

16.1 Flash Memory Organisation and Types

Flash memory is partitioned into sectors. The number of sectors depends on the Flash device type, 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 tables provide details of the on-chip Flash memory and supported external Flash devices for the JN517x family of microcontrollers.

JN517x Chip	Number of Sectors	Sector Size (Kbytes)	Total Size (Kbytes)
JN5179	16	32	512
JN5178	8	32	256
JN5174	5	32	160

Table 8: On-chip Flash Memory

Manufacturer	Flash Device	Number of Sectors	Sector Size (Kbytes)	Total Size (Kbytes)
Atmel	AT25F512	2	32	64
STMicroelectronics	M25P05A	2	32	64
Microchip	SST25VF010A	4	32	128
STMicroelectronics	M25P10A	4	32	128
STMicroelectronics	M25P20	4	64	256
Winbond	W25X20B	4	64	256
STMicroelectronics	M25P40	8	64	512

Table 9: Supported External Flash Devices

16.2 API Functions

The supplied Flash Memory functions can be used to interact with the on-chip Flash device and any compatible external Flash device (detailed in Section 16.1). The functions are able to access any sector of Flash memory - the application is stored from the first sector (0) and application data is normally stored in the final sector - you should refer to the data sheet for the Flash device to obtain the necessary sector details. The Flash Memory functions are fully detailed in Chapter 32.

16.3 Operating on Flash Memory

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

The first Flash memory function called must be the initialisation function **bAHI_FlashInit()**. In the case of external Flash memory, this function requires the attached Flash device type to be specified.

> **Note 1:** If you wish to use both internal (on-chip) and external Flash memory devices, you will need to call **bAHI_FlashInit()** when switching between them.
>
> **Note 2:** All Flash addresses specified in the Flash Memory functions are offsets from the start of Flash memory and not absolute addresses.
>
> **Note 3:** The **bAHI_FlashEECerrorInterruptSet()** function can be used to enable interrupts that are generated when an error occurs in the on-chip Flash device. A user-defined callback function is also registered which is invoked when a Flash memory interrupt occurs.
>
> **Note 4:** If you are not using one of the supported external Flash memory devices (see Table 9 on page 125), you will need to supply a set of custom Flash functions that will be used by the supplied Flash functions to access your Flash memory device. These custom functions must be registered in the call to **bAHI_FlashInit()** - for more information, refer to the description of this function in Chapter 32.

16.3.1 Erasing Data from Flash Memory

Erasing a portion of Flash memory involves setting any 0 bits to 1. The function **bAHI_FlashEraseSector()** can be used to erase an entire sector of Flash memory. Any sector can be erased.

> ***Caution:*** *Be careful not to erase essential data such as the application code. The application is stored from the start of the on-chip Flash memory (starting in Sector 0).*

> **Note:** The internal Flash memory of the JN517x device has a sector-erase time of approximately 100ms.

16.3.2 Reading Data from Flash Memory

The function **bAHI_FullFlashRead()** can be used to read data from any sector of Flash memory. This function can be used to read a portion of data starting at any point within the sector.

16.3.3 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 16.3.1, before writing the first data to it.

The function **bAHI_FullFlashProgram()** can be used to write data to any sector of Flash memory. This function can be used to write a portion of data containing a multiple of 16 bytes starting on a 16-byte boundary 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).

One way to ensure that data is added successfully to a sector is as follows:

1. Read the entire sector into RAM (see Section 16.3.2).
2. Erase the entire sector in Flash memory (see Section 16.3.1).
3. Add the new data to the existing data in RAM.
4. Write all of this data back to the sector in Flash memory.

> ***Caution 1:*** *Each sector of the internal Flash memory in the JN517x device is divided into 16-byte pagewords. A write to a non-blank pageword must not be performed - the sector containing the non-blank pageword should first be erased using* ***bAHI_FlashEraseSector()*** *before writing to the pageword. If the user omits the sector-erase operation, a subsequent error will likely result when reading from the pageword - this read-error will trigger an interrupt and execute the callback function registered using* ***bAHI_FlashEECerrorInterruptSet()****.*
>
> ***Caution 2:*** *The internal Flash memory of the JN517x device has an endurance limit of 10000 write/erase cycles per sector. Refer to the device-specific data sheet for the endurance limit of the external Flash memory.*

> **Note:** The internal Flash memory of the JN517x device has a sector write-time of approximately 1ms.

16.4 Controlling Power to External Flash Memory

Any external Flash memory can be optionally powered off while the JN517x microcontroller is in a sleep mode (including Deep Sleep). An unpowered Flash device during sleep allows greater power savings and extends battery life.

Two functions (see below) are provided for controlling power to an external Flash device, but these are only applicable to the following STMicroelectronics devices:

- STM25P05A
- STM25P10A
- STM25P20
- STM25P40

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

> ***Caution:*** *These functions must not be called when using JN517x on-chip Flash memory device (selected in* ***bAHI_FlashInit()****).*

The necessary function calls before and after sleep are outlined as follows.

Before Sleep

The above external 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.

After Sleep

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

Tip: In order to conserve power, you may wish to power down the external Flash memory device at JN517x start-up and only power up the Flash device when required.

17. EEPROM

This chapter describes access to the JN517x on-chip EEPROM using functions of the Integrated Peripherals API. This non-volatile memory is used to store data that must be preserved while the JN517x device is not powered or during sleep without RAM held. EEPROM which is blank (no data) comprises entirely of binary 0s. When data is written to it, the relevant bits are changed from 0 to 1. Functions are provided for writing to, reading from and erasing the EEPROM.

Although the functions referenced in this chapter provide direct access to the EEPROM device, *it is recommended that they are not used or are used with caution*, for the following reasons:

- ZigBee nodes use the Persistent Data Manager (PDM) which also accesses the EEPROM. PDM is part of the JN51xx Core Utilities (JCU), supplied in the JN517x ZigBee SDKs, and is described in the *JN51xx Core Utilities User Guide (JN-UG-3116)*. The advantages of using PDM include:
    - 'Wear levelling' to achieve the uniform use of the EEPROM
    - Record IDs to avoid the use of memory addresses

  If PDM and the EEPROM direct-access function set are both to be used, it is important to avoid conflicts between the two - therefore, they must never access the same part of the EEPROM.
- ZigBee RF4CE uses the EEPROM direct-access functions itself so, again, conflicts must be avoided.

17.1 Initialisation

In order to access the EEPROM from the application, the initialisation function **u16AHI_InitialiseEEP()** must first be called.

The EEPROM is organised in terms of segments of 64 bytes each, and the above function returns the following information about the available segments:

- Number of segments
- Number of bytes in each segment

The segments are indexed from 0.

> **Note:** The final segment of EEPROM is reserved for production data and cannot be written to or erased.

17.2 Writing to the EEPROM

A block of data can be written to a specified EEPROM segment using the function **iAHI_WriteDataIntoEEPROMsegment()**. The data can be written starting at any (byte) offset from the beginning of the segment. The function will not allow a segment to overflow - if the length of the data block to be written is greater than the memory space up to the end of the segment, the function will return an error and will not write any data.

17.3 Reading from the EEPROM

A block of data can be read from a specified EEPROM segment using the function **iAHI_ReadDataFromEEPROMsegment()**. The data can be read starting at any (byte) offset from the beginning of the segment. If the length of the data block to be read is greater than the memory space up to the end of the segment, the function will return an error and will not read any data.

17.4 Erasing the EEPROM

The EEPROM can be erased a whole segment at a time. The function **iAHI_EraseEEPROMsegment()** can be used to erase a specified segment.

Part II:
Reference Information

18. General Functions

This chapter describes various functions of the Integrated Peripherals API that are not associated with any of the main peripheral blocks on a JN517x microcontroller.

The functions in this chapter include:

- API initialisation function
- Functions to implement antenna diversity
- Functions to control the random number generator
- Processor stack overflow function
- Functions for accessing on-chip Non-Volatile Memory
- Functions for preserving debug information during sleep
- Function for reducing the maximum receive power on JN5169
- Function for applying Wi-Fi counter-measures on JN517x
- Function for configuring the JN517x device to work with a specific module

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

Note: For guidance on using these functions in JN517x application code, refer to Chapter 2.

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

u32AHI_Init

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

Description

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

Note: This function must be called before initialising the Application Queue API (if used). For more information on the latter, refer to the *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*.

Parameters

None

Returns

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

vAHI_RfPinOutputEnable

```
void vAHI_RfPinOutputEnable(
                    teModuleRfPin eModuleRfPinMask);
```

Description

This function configures DIO pins as RF state outputs, either RFRX (high when radio is in receive mode) or RFTX (high when radio is in transmit mode). The function parameter is a bitmap of the various options. RFTX can be assigned to DIO1 or DIO3, and RFRX can be assigned to DIO0 or DIO2. It is recommended that only one of the alternatives for each of RFTX and RFRX are selected at a time.

Note that the same functionality can be achieved by configuring each DIO pin individually using the **vAHI_SetDIOpinMultiplexValue()** function.

Parameters

eModuleRfPinMask	Bitmap of RFRX and RFTX options on the DIO0-3 pins, achieved by logical OR of one or more of the following enumerated values: E_MODULE_RFRX_DIO0 (RFRX on DIO0) E_MODULE_RFTX_DIO1 (RFTX on DIO1) E_MODULE_RFRX_DIO2 (RFRX on DIO2) E_MODULE_RFTX_DIO3 (RFTX on DIO3)

Returns

None

vAHI_AntennaDiversityEnable

```
void vAHI_AntennaDiversityEnable(
                                  bool_t bRxDiversity,
                                  bool_t bTxDiversity);
```

Description

This function can be used to independently enable/disable antenna diversity on the transmit and receive paths. The use of antenna diversity requires two antennas to be connected to the JN517x device via a switch controlled by two signals. The signals are routed to DIO pins using the function **vAHI_SetDIOpinMultiplexValue()**.

Parameters

bRxDiversity	Enable/disable antenna diversity on receive path: TRUE - enable FALSE - disable
bTxDiversity	Enable/disable antenna diversity on transmit path: TRUE - enable FALSE - disable

Returns

None

u8AHI_AntennaDiversityStatus

```
uint8 u8AHI_AntennaDiversityStatus(void);
```

Description

This function can be used to obtain the latest antenna diversity status (when two antennae are connected to the JN517x device and antenna diversity has been enabled through a call to **vAHI_AntennaDiversityEnable()**). The use of antenna diversity requires two antennas to be connected to the JN517x device via a switch controlled by two signals - these signals are routed to DIO pins by calling the function **vAHI_SetDIOpinMultiplexValue()**.

The function returns a bitmap containing the following information:

- Antenna used for the last transmit (bit 0)
- Antenna used for the last receive (bit 1)
- Currently selected antenna (bit 2)

For each of the bits above, 0 corresponds to signal ADE and 1 corresponds to ADO. The state of the other antenna control signal will be the complement of this value.

Parameters

None

Returns

Result is a bitmap which can be bitwise ANDed with the following masks:

E_AHI_ANTDIV_STAT_TX_MASK (0x1) - extracts antenna used for last Tx

E_AHI_ANTDIV_STAT_RX_MASK (0x2) - extracts antenna used for last Rx

E_AHI_ANTDIV_STAT_ANT_MASK (0x4) - extracts antenna currently selected

vAHI_AntennaDiversityControl

```
void vAHI_AntennaDiversityControl(
                    uint8 u8RxRssiThreshold,
                    uint8 u8RxCorrThreshold);
```

Description

This function can be used for application control of antenna diversity (enabled through a call to **vAHI_AntennaDiversityEnable()**), in the following ways:

- Receive diversity RSSI threshold can be set which determines the minimum acceptable receive signal strength below which the antenna may be switched (also subject to other conditions - see Section 2.4)
- Receive diversity Correlation threshold can be set which determines the minimum acceptable receive signal quality below which the antenna may be switched (also subject to other conditions - see Section 2.4)

Parameters

u8RxRssiThreshold	Receive diversity RSSI threshold, in 1dB steps from 0 to 31 (default value is 25 - it not recommended to use values less than 25)
u8RxCorrThreshold	Receive diversity Correlation threshold, from 0 to 63 (default value is 25 - it is not recommended to use values less than 25 or greater than 40)

Returns

None

vAHI_AntennaDiversitySwitch

```
void vAHI_AntennaDiversitySwitch(void);
```

Description

This function can be used by an application to manually switch the currently selected antenna for the control of antenna diversity. Note that calling this function will generally not be required because it is expected that most applications will make use of the automatic transmit and/or receive antenna diversity control features that are enabled by calling **vAHI_AntennaDiversityEnable()**.

The use of antenna diversity requires two antennas to be connected to the JN517x device via a switch controlled by two signals - these signals are routed to DIO pins by calling the function **vAHI_SetDIOpinMultiplexValue()**.

Parameters

None

Returns

None

vAHI_StartRandomNumberGenerator

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

Description

This function starts the random number generator on the JN517x device, which produces 16-bit random values. The generator can be started in one of two modes:

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

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

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

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

Note that the random number generator uses the 32kHz clock domain (see Section 3.1) and will not operate properly if a high-precision external 32kHz clock source is used. Therefore, if generating random numbers in your application, you are advised to use the internal RC oscillator or a low-precision external clock source.

Parameters

bMode	Generator mode: E_AHI_RND_SINGLE_SHOT (single-shot mode) E_AHI_RND_CONTINUOUS (continuous mode)
bIntEn	Enable/disable interrupts setting: E_AHI_INTS_ENABLED(enable) E_AHI_INTS_DISABLED(disable)

Returns

None

vAHI_StopRandomNumberGenerator

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

Description

This function stops the random number generator on the JN517x device, if it has been started in continuous mode using **vAHI_StartRandomNumberGenerator()**.

Parameters

None

Returns

None

u16AHI_ReadRandomNumber

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

Description

This function obtains the last 16-bit random value produced by the random number generator on the JN517x device. The function can only be called once the random number generator has generated a new random number.

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

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

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

Parameters

None

Returns

16-bit random integer

bAHI_RndNumPoll

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

Description

This function can be used to poll the random number generator on the JN517x device - that is, to determine whether the generator has produced a new random value.

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

Parameters

None

Returns

Availability of new random value, one of:
TRUE - random value available
FALSE - no random value available

vAHI_WriteNVData

```
void vAHI_WriteNVData(uint8 u8Location,
                      uint32 u32WriteData);
```

Description

This function writes the specified 32-bit word to the specified location in the JN517x internal 4-word NVM (Non-Volatile Memory). The JN517x internal NVM contains four 32-bit locations, numbered 0 to 3.

Parameters

u8Location	Number of NVM location to which word is to be written: 0, 1, 2 or 3
u32WriteData	32-bit word to be written to NVM

Returns

None

u32AHI_ReadNVData

```
uint32 u32AHI_ReadNVData(uint8 u8Location);
```

Description

This function reads the 32-bit word from the specified location in the JN517x internal 4-word NVM (Non-Volatile Memory). The JN517x internal NVM contains four 32-bit locations, numbered 0 to 3.

Parameters

u8Location	Number of NVM location from which word is to be read: 0, 1, 2 or 3

Returns

32-bit word read from NVM

vAHI_RadioSetReducedInputPower

```
void vAHI_RadioSetReducedInputPower(bool_t bReduced);
```

Description

This function can be used to reduce the maximum radio signal power that the JN517x device can receive before saturating. The function can be called at any time.

The JN517x device can receive radio signals of up to 10 dBm before the input is saturated. However, this function can be used to configure the device to saturate at a lower incoming signal level of 0 dBm, which has the advantage of drawing less current and prolonging battery life.

The function has a Boolean parameter for which TRUE selects the reduced maximum input level and FALSE selects the normal (default) input level.

Parameters

bReduced Enable/disable reduced maximum input power:
TRUE - Enable reduced saturation level
FALSE - Disable reduced saturation level (restore default level)

Returns

None

vAHI_ModuleConfigure

```
void vAHI_ModuleConfigure(teModule eModule);
```

Description

This function can be used to configure a JN517x device to operate on a particular JN517x module type. If used, it must be called before initialising the IEEE802.15.4-based protocol stack (e.g. ZigBee PRO stack).

Depending on the module type specified through *eModule*, the function configures:

- Transmit power limits to ensure that the device remains within compliance limits
- CCA (Clear Channel Assessment) threshold to match the LNA (Low-Noise Amplifier) on the receive path
- DIO used to drive the PA (Power Amplifier) and LNA

Note that the function does not include the configuration of antenna diversity or LNA bypass mode (the latter is available on the JN5179-001-M16 module only). However, there is a special value of *eModule* for use with LNA bypass mode because the CCA threshold is specific to this mode of operation.

Important: This function is provided as source code only and can be customised as required. For more information on this function and how to use it, refer to Section 2.6.

Parameters

eModule Enumeration indicating the type of JN517x module (see Section 2.6.2), one of:

E_MODULE_JN5179_001_M10_ETSI
E_MODULE_JN5179_001_M13_ETSI
E_MODULE_JN5179_001_M10_FCC
E_MODULE_JN5179_001_M13_FCC
E_MODULE_JN5179_001_M16_FCC
E_MODULE_JN5179_001_M16_FCC_LNA_BYPASS

Returns

None

19. System Controller Functions

This chapter describes the functions that interface to the System Controller on the JN517x microcontroller.

The functions detailed in this chapter cover the following areas:

- Power management
- Clock management
- Supply voltage monitoring (Voltage brownout)
- Chip reset

Note: For information on the above chip features and guidance on using the System Controller functions in JN517x application code, refer to Chapter 3.

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

u16AHI_PowerStatus

```
uint16 u16AHI_PowerStatus(void);
```

Description

This function returns power domain status information for the JN517x microcontroller - in particular, whether:

- Device has completed a sleep-wake cycle
- RAM contents were retained during sleep
- Analogue power domain is switched on
- Protocol logic is operational (clock is enabled)
- Watchdog timeout was responsible for the last device restart
- 32kHz clock is ready (e.g. following a reset or wake-up)
- Device has just come out of Deep Sleep mode (rather than a reset)

Note that you must check whether the 32kHz clock is ready before starting a wake timer.

Parameters

None

Returns

Returns the power domain status information in bits 0-3, 7 and 10-11 of the 16-bit return value:

Bit	Reads a '1' if...
0	Device has completed a sleep-wake cycle
1	RAM contents were retained during sleep
2	Analogue power domain is switched on
3	Protocol logic is operational
4-6	Unused
7	Watchdog caused last device restart
8-9	Unused
10	32kHz clock is ready
11	Device has just come out of Deep Sleep mode
12-15	Unused

vAHI_CpuDoze

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

Description

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

The function returns when the CPU re-starts.

Parameters

None

Returns

None

vAHI_Sleep

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

Description

This function puts the JN517x device into Sleep mode, being one of four 'normal' Sleep modes or Deep Sleep mode. The normal sleep modes are distinguished by whether on-chip RAM remains powered and whether the 32kHz oscillator is left running during sleep (see parameter description below).

Note 1: If an external source is used for the 32kHz oscillator on the JN517x device (see page 151), it is not recommended that the oscillator is stopped on entering Sleep mode.

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

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

  External Flash memory is not powered down during normal sleep mode. If required, you can power down the Flash memory device using the function **vAHI_FlashPowerDown()**, which must be called before **vAHI_Sleep()**, provided you are using a compatible Flash memory device - refer to the description of **vAHI_FlashPowerDown()** on page 418.
- In Deep Sleep mode, all components of the chip are powered down and the device can only be woken by the device's reset line being pulled low or an external event which triggers a change on a DIO pin (the relevant DIO must be configured as an input and DIO interrupts must be enabled).

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

Parameters

sSleepMode	Required Sleep mode, one of: E_AHI_SLEEP_OSCON_RAMON 32kHz oscillator on and RAM on (warm restart) E_AHI_SLEEP_OSCON_RAMOFF 32kHz oscillator on and RAM off (cold restart) E_AHI_SLEEP_OSCOFF_RAMON 32kHz oscillator off and RAM on (warm restart) E_AHI_SLEEP_OSCOFF_RAMOFF 32kHz oscillator off and RAM off (cold restart) E_AHI_SLEEP_DEEP Deep Sleep (similar to OSCOFF_RAMOFF - cold restart)

Returns

None

vAHI_ProtocolPower

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

Description

This function is used to enable or disable the clock to the wireless transceiver - the clock is simply disabled (gated) while the domain remains powered.

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

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

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

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

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

Parameters

bOnNotOff	Setting for clock to wireless transceiver: TRUE to switch the clock ON FALSE to switch the clock OFF

Returns

None

bAHI_Set32KhzClockMode

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

Description

This function selects an external source for the 32kHz clock for the JN517x device (the function is used to move from the internal source to an external source). The selected clock can be either of the following options:

- **External module (RC circuit):** This clock must be supplied on DIO7
- **External crystal:** This circuit must be attached on DIO7 and DIO8

If the external crystal is selected and is not already running, it will be started and this function will not return until the crystal has stabilised (which can take up to 1 second). If the clock fails to start after 8 seconds, the device switches back to the internal RC oscillator.

If this function or **vAHI_Init32KhzXtal()** is not called, the internal 32kHz RC oscillator is used by default. Note that once an external 32kHz clock source has been selected using this function, it is not possible to switch back to the internal RC oscillator.

If required, this function should be called near the start of the application. In particular, if selecting the external crystal, the function must be called before Timer 0 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 there is no need to explicitly configure DIO7 or DIO8 as an input, as this is done automatically by the function.

When selecting an external module, you must disable the pull-up on DIO7 using the function **vAHI_DioSetPullup()**. However, when selecting the external crystal, the pull-ups on DIO7 and DIO8 are disabled automatically.

Parameters

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

Returns

TRUE - external clock is running
FALSE - external clock has not started and the internal RC oscillator is being used

vAHI_Init32KhzXtal

```
void vAHI_Init32KhzXtal(void);
```

Description

This function starts an external crystal and switches the 32kHz clock source for the JN517x device to it.

The external crystal must be connected to the device via DIO7 and DIO8. There is no need to explicitly configure DIO7 or DIO8 as an input, as this is done automatically by the function.

The external crystal that has been started needs time to stabilise before it can be used as a clock source. The function returns immediately before the clock stabilises and the application can perform other processing or put the JN517x device into sleep mode while waiting for the crystal to become ready - it takes up to 1 second to stabilise. In the case of sleep, the application should typically set a wake timer to wake the device after 1 second.

If this function or **bAHI_Set32KhzClockMode()** is not called, the internal 32kHz RC oscillator is used by default. Note that once an external 32kHz clock source has been selected using this function, it is not possible to switch back to the internal RC oscillator.

Parameters

None

Returns

None

vAHI_Trim32KhzRC

void vAHI_Trim32KhzRC(uint8 *u8Value***);**

Description

This function sets the electrical current consumption of the 32kHz RC oscillator (external module), which determines the accuracy of the clock frequency produced - the higher the current, the more accurate the generated clock frequency.

Presently, two current settings are available; 0.53µA and 0.35µA, with corresponding frequency calibration errors of ±300ppm and ±600ppm, respectively.

Parameters

u8Value	Current consumption to be set: 0: Reserved 1: Reserved 2: 0.53µA (default) 3: 0.35µA 4-7: No effect

Returns

None

vAHI_SelectClockSource

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

Description

This function selects the clock source for the system clock on the JN517x device. The clock options are:

- Crystal oscillator (XTAL) of frequency 32MHz, derived from external crystal
- Internal high-speed RC oscillator of frequency 27MHz (uncalibrated), but can be adjusted to 32MHz (calibrated) using the function **bAHI_TrimHighSpeedRCOsc()**

If used, the external crystal is connected to pins XTAL_IN and XTAL_OUT.

The CPU clock and peripheral clock are divided down versions of this clock source. The CPU clock divisor is controlled using the function **bAHI_SetClockRate()**. The peripheral clock is produced by dividing this clock source by two. Thus, the crystal oscillator will produce a 16MHz peripheral clock and the RC oscillator will produce a peripheral clock of 13.5MHz (±18%, uncalibrated) or 16MHz (±5% calibrated).

> ***Caution:*** *You will not be able to run the full system while using the RC oscillator. It is possible to execute code while using this clock source, but it is not possible to transmit or receive. Further, timing intervals for the timers may need to be based on a frequency of 13.5MHz.*

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

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

Parameters

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

Returns

None

bAHI_GetClkSource

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

Description

This function obtains the identity of the clock source for the system clock on the JN517x device. The clock options are:

- Crystal oscillator (XTAL) of frequency 32MHz, derived from external crystal
- Internal high-speed RC oscillator of frequency 27MHz (uncalibrated), but can be adjusted to 32MHz (calibrated) using the function **bAHI_TrimHighSpeedRCOsc()**

If the high-speed RC oscillator is the system clock source, **bAHI_GetClkSource()** does not indicate the operating frequency of the oscillator.

Parameters

None

Returns

Clock source, one of:
TRUE - RC oscillator
FALSE - Crystal oscillator

bAHI_SetClockRate

bool_t bAHI_SetClockRate(uint8 *u8Speed***);**

Description

This function is used to select a CPU clock rate on the JN517x device by setting the divisor used to derive the CPU clock from the system source clock.

The system clock source is selected using the function **vAHI_SelectClockSource()** as one of:

- 32MHz external crystal oscillator
- High-speed internal RC oscillator of frequency 27MHz (uncalibrated), but can be adjusted to 32MHz (calibrated) using the function **bAHI_TrimHighSpeedRCOsc()**

The possible divisors are 1, 2, 4, 8, 16 and 32.

Irrespective of the setting made with this function, the CPU clock rate will default to 16MHz or 13.5MHz (clock divisor of 2) following sleep - that is, the clock divisor configured before sleep is not automatically re-applied after sleep.

Parameters

u8Speed Divisor for desired CPU clock frequency:

u8Speed	**Clock Divisor**	**Resulting Frequency (MHz)**	
		From 32MHz	**From 27MHz**
000	8	4	3.38
001	4	8	6.75
010	2	16	13.5
011	1	32	27
100	Invalid		
101			
110	16	2	1.69
111	32	1	0.84

Note: When the RC oscillator is used as the source, the resulting CPU clock frequency is dictated by the actual RC oscillator frequency, which can be 27MHz (±18%) or 32MHz (±5% when calibrated).

Returns

TRUE - successful
FALSE - invalid divisor value specified

u8AHI_GetSystemClkRate

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

Description

This function obtains the divisor used to divide down the source clock to produce the CPU clock on the JN517x device.

The system clock source is selected using the function **vAHI_SelectClockSource()** as one of:

- 32MHz external crystal oscillator
- High-speed internal RC oscillator of frequency 27MHz (uncalibrated), but can be adjusted to 32MHz using the function **bAHI_TrimHighSpeedRCOsc()**

The current clock source can be obtained using the function **bAHI_GetClkSource()**, but this function does not indicate the operating frequency of the RC oscillator (if used).

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

The possible divisors are 1, 2, 4, 8, 16 and 32. The CPU clock frequency can be calculated by dividing the source clock frequency by the divisor returned by this function. The results are summarised in the table below.

<table>
<tr><th rowspan="2">Returned Value</th><th rowspan="2">Clock Divisor</th><th colspan="2">Resulting Frequency (MHz)</th></tr>
<tr><th>From 32MHz</th><th>From 27MHz</th></tr>
<tr><td>0</td><td>8</td><td>4</td><td>3.38</td></tr>
<tr><td>1</td><td>4</td><td>8</td><td>6.75</td></tr>
<tr><td>2</td><td>2</td><td>16</td><td>13.5</td></tr>
<tr><td>3</td><td>1</td><td>32</td><td>27</td></tr>
<tr><td>4</td><td rowspan="2" colspan="3">Invalid</td></tr>
<tr><td>5</td></tr>
<tr><td>6</td><td>16</td><td>2</td><td>1.69</td></tr>
<tr><td>7</td><td>32</td><td>1</td><td>0.84</td></tr>
</table>

Note: When the RC oscillator is used as the source, the resulting system clock frequency is dictated by the actual RC oscillator frequency, which can be 27MHz (±18%) or 32MHz (±5% when calibrated).

Parameters

None

Returns

0: Divisor of 8

1: Divisor of 4

2: Divisor of 2

3: Divisor of 1 (source frequency untouched)

6: Divisor of 16

7: Divisor of 32

bAHI_Clock32MHzStable

```
bool_t bAHI_Clock32MHzStable(void);
```

Description

This function can be used to check whether the 32MHz crystal oscillator (sourced externally) is running and stable.

Parameters

None

Returns

TRUE - oscillator is stable

FALSE - oscillator is not stable

vAHI_ClockXtalPull

void vAHI_ClockXtalPull(uint8 *u8PullValue***);**

Description

This function can be used to decrease (pull) the frequency of the 32MHz crystal oscillator by increasing the crystal load capacitance in the oscillator tuning circuit. If the JN517x device operates at temperatures in excess of 85°C, it may be necessary to call this function to maintain the frequency tolerance of the clock within the 40ppm limit specified by the IEEE 802.15.4 standard.

The crystal pulling coefficient specifies the sensitivity of the crystal frequency with respect to the crystal load capacitance. Crystals suitable for use with the JN517x will typically have a crystal pulling coefficient value in the range of 15 to 25 ppm/pF. Although the crystal pulling coefficient has a positive value, it should be noted that the crystal frequency will decrease with increasing crystal load capacitance.

The formula for calculating the crystal pulling coefficient *(Δf)* is given by:

$$\Delta f = \frac{C_m \times 10^6}{2 \times (C_L + C_S)^2} \qquad ppm/F$$

where,

C_m is the crystal motional capacitance (e.g. 4.4pF)
C_L is the crystal load capacitance (e.g. 9pF)
C_S is the crystal shunt or package capacitance (e.g. 1pF)

The example crystal capacitance values quoted above yield a crystal pulling coefficient of 22ppm/pF. Therefore, an increase of the crystal load capacitance (C_L) by 1pF will reduce the crystal oscillating frequency by 22ppm.

Note: Please refer to the JN517x Data Sheet and the crystal manufacturer data sheet for specific details of the crystal capacitances. Also refer to the Application Note *JN516x/7x Temperature-dependent Operating Guidelines (JN-AN-1186)* for details of the crystal oscillator frequency compensation over temperature.

Parameters

u8PullValue Pull-value controls the additional crystal load capacitance:

0: No additional crystal load capacitance (default)
1: 1pF
2: 2pF
3: 3pF

Returns

None

vAHI_Trim32Mhz

```
void vAHI_Trim32Mhz(uint8 u8Value);
```

Description

This function sets the bias current of the 32MHz crystal oscillator in the main amplifier, which determines the accuracy of the clock frequency produced.

Parameters

u8Value	Current consumption to be set: 000: 243µA 001: 212µA 010: 182µA 011: 152µA 100: 768µA 101: 700µA 110: 362µA 111: 303µA

Returns

None

vAHI_PowerXTAL

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

Description

This function can be used to configure whether the 32MHz crystal oscillator is powered down (to minimise power usage) when the High Speed RC Oscillator is being used as the system clock source, with no automatic switch-over to the 32MHz oscillator.

> **Note:** This function only has an effect when selecting a clock source. It does not provide dynamic control during operation.

Parameters

bIsOn	Power setting for the 32MHz crystal oscillator: TRUE - powered down FALSE - powered

Returns

None

vAHI_EnableFastStartUp

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

Description

This function can be used to modify the (default) fast start-up following sleep. If required, the function must be called before entering sleep mode.

The external 32MHz crystal oscillator is powered down during sleep and takes some time to become available again when the JN517x device wakes. A more rapid start-up from sleep can be achieved by using the internal high-speed 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. This rapid start-up following sleep occurs automatically by default.

This function can be used to configure the switch to the crystal oscillator to be either automatic or manual (selected through the *bMode* parameter):

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

It is also possible to use this function to configure the device to keep the RC oscillator as the source for the system clock when re-starting from sleep. To do this, it is necessary to select a manual switch (through the *bMode* parameter) but not perform any switch.

While the internal high-speed RC oscillator is being used, you should not attempt to transmit or receive, and you can only use the JN517x peripherals with special care - see Section 3.1.3.

To conserve power, you can use the *bPowerDown* parameter to keep the crystal oscillator powered down until it is needed.

Parameters

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

Returns

None

bAHI_TrimHighSpeedRCOsc

```
bool_t bAHI_TrimHighSpeedRCOsc(void);
```

Description

This function can be used on the JN517x device to adjust the frequency of the internal high-speed RC oscillator from 27MHz uncalibrated to 32MHz calibrated.

Parameters

None

Returns

TRUE - RC oscillator frequency successfully changed

FALSE - Unable to change RC oscillator frequency

vAHI_OptimiseWaitStates

```
void vAHI_OptimiseWaitStates(void);
```

Description

This function recalculates the wait-state settings for the internal Flash memory and EEPROM devices after the system clock source or CPU clock frequency has been changed to minimise the Flash access time. The function is automatically called after calling **vAHI_SelectClockSource()** or **bAHI_SetClockRate()** but should preferably be called by the application in either of the following circumstances:

- at the start of an application (cold start or warm restart) with the system clock running from the internal high-speed RC oscillator
- after switching from the internal high-speed RC oscillator to the external 32MHz crystal

Note: By default, following a reset or on waking from sleep, the device will automatically switch from using the internal high-speed RC oscillator to the external 32MHz crystal as the system clock source once the crystal oscillator has stabilised.

Parameters

None

Returns

None

vAHI_BrownOutConfigure

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

Description

This function configures and enables the Supply Voltage Monitor (SVM), which can be used to detect a brownout condition on the JN517x device.

Brownout is the point at which the chip supply voltage falls to (or below) a pre-defined level. The default brownout level is set to 2.0 V in the JN517x device during manufacture. This function can be used to temporarily over-ride the default brownout voltage with one of several voltage levels. Before the new setting takes effect, there is a delay of up to 3.3µs.

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

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

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

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

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

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

Parameters

u8VboSelect	Voltage threshold for brownout: E_AHI_VBOREF_1V95 - 0 (1.95V) E_AHI_VBOREF_2V0 - 1 (2.0V, default) E_AHI_VBOREF_2V1 - 2 (2.1V) E_AHI_VBOREF_2V2 - 3 (2.2V) E_AHI_VBOREF_2V3 - 4 (2.3V) E_AHI_VBOREF_2V4 - 5 (2.4V) E_AHI_VBOREF_2V7 - 6 (2.7V) E_AHI_VBOREF_3V0 - 7 (3.0V)
bVboRestEn	Enable/disable 'reset on brownout': TRUE to enable reset FALSE to disable reset
bVboEn	Enable/disable SVM: TRUE to enable SVM FALSE to disable SVM
bVboIntEnFalling	Enable/disable interrupt generated when the brownout bit falls, indicating that the device has come out of the brownout state: TRUE to enable interrupt FALSE to disable interrupt
bVboIntEnRising	Enable/disable interrupt generated when the brownout bit rises, indicating that the device has entered the brownout state: TRUE to enable interrupt FALSE to disable interrupt

Returns

None

bAHI_BrownOutStatus

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

Description

This function can be used to check whether the current supply voltage to the JN517x device is above or below the brownout voltage setting (the default value or the value configured using the function **vAHI_BrownOutConfigure()**).

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

There may be a delay before **bAHI_BrownOutStatus()** returns, if the brownout configuration has recently changed - this delay is up to 3.3µs.

Parameters

None

Returns

TRUE - supply voltage is below brownout voltage

FALSE - supply voltage is above brownout voltage

bAHI_BrownOutEventResetStatus

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

Description

This function can be called following a JN517x device reset to determine whether the reset event was caused by a brownout. This allows the application to then take any necessary action following a confirmed brownout.

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

Parameters

None

Returns

TRUE if brownout caused reset, FALSE otherwise

u32AHI_BrownOutPoll

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

Description

This function can be used to poll for a brownout on the JN517x device - that is, to check whether a brownout has occurred. The returned value will indicate whether the chip supply voltage has fallen below or risen above the brownout voltage (or both). Polling using this function clears the brownout status, so that a new and valid result will be obtained the next time the function is called.

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

Parameters

None

Returns

32-bit value containing brownout status:

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

vAHI_SwReset

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

Description

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

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

Parameters

None

Returns

None

vAHI_SetJTAGdebugger

```
void vAHI_SetJTAGdebugger(bool_t bEnable,
                          uint8 u8TDOpin,
                          uint8 u8TDIpin,
                          uint8 u8TCKpin,
                          unit8 u8TMSpin);
```

Description

This function can be used to enable or disable the JTAG debugger hardware, and to select the DIO pins on which the JTAG signals will be located. The pin locations specified allows DIO usage conflicts between the JTAG debugger and any enabled peripheral to be more easily avoided.

Note 1: This function will typically not be required in an application because the debugger will be automatically configured by the bootloader depending on makefile build options (see below).

Note 2: The bootloader will automatically enable the debugger hardware if the makefile build option variable HARDWARE_DEBUG_ENABLED is set to 1. The bootloader will also configure the DIO pins for the enabled debugger as directed by the makefile build option variable DEBUG_PORT.

Parameters

bEnable	Enable or disable debugger: TRUE - enable FALSE - disable
u8TDOpin	DIO on which TDO JTAG signal is located: 9 or 15
u8TDIpin	DIO on which TDI JTAG signal is located: 7 or 10
u8TCKpin	DIO on which TCK JTAG signal is located: 6 or 17
u8TMSpin	DIO on which TMS JTAG signal is located: 11 or 18

Returns

None

vAHI_ClearSystemEventStatus

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

Description

This function clears the specified System Controller interrupt sources on a JN517x device. A bitmask indicating the interrupt sources to be cleared must be passed into the function.

Parameters

u32BitMask	Bitmask of the System Controller interrupt sources to be cleared. To clear an interrupt, the corresponding bit must be set to 1 - for bit numbers, refer to Table 13 on page 437

Returns

None

vAHI_SysCtrlRegisterCallback

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

Description

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

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId*, **uint32** *u32ItemBitmap***);**

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

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

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

prSysCtrlCallback	Pointer to callback function to be registered

Returns

None

20. Analogue Peripheral Functions

This chapter describes the functions that are used to control the analogue peripherals of the JN517x microcontroller. These are the on-chip peripheral types with analogue inputs or outputs: Analogue-to-Digital Converter (ADC) and comparator.

The analogue peripheral functions are divided into the following sections:

- Common analogue peripheral functions, described in Section 20.1
- ADC functions, described in Section 20.2
- ADC with DMA Engine functions, described in Section 20.3
- Comparator functions, described in Section 20.4

> **Note:** For information on the analogue peripherals and guidance on using these functions in JN517x application code, refer to Chapter 4.

20.1 Common Analogue Peripheral Functions

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

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

vAHI_ApConfigure

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

Description

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

- The regulator used to power the analogue peripherals must be enabled for the remaining input parameters to take effect and for the ADC to operate. The regulator minimises digital noise and is sourced from the analogue supply pin VDD1.
- Interrupts can be enabled that are generated after each ADC conversion.
- The divisor is specified to obtain the ADC clock from the peripheral clock.
- The 'sampling interval' is specified as a number of clock periods.
- The source of the reference voltage, V_{ref}, is specified.

The peripheral clock runs at 16MHz when the system clock is sourced from the external 32MHz crystal. The supplied clock divisor enumerations (see the parameter *u8ClockDivRatio* below) for producing the ADC clock are based on a 16MHz peripheral clock. If the peripheral clock frequency is not exactly 16MHz, the resultant ADC clock frequency will be scaled accordingly.

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

[(3 x *sampling interval*) + 13] x *clock period*

Parameters

bAPRegulator	Enable/disable the regulator used to power the analogue peripherals: E_AHI_AP_REGULATOR_ENABLE E_AHI_AP_REGULATOR_DISABLE
bIntEnable	Enable/disable interrupt when ADC conversion completes: E_AHI_AP_INT_ENABLE E_AHI_AP_INT_DISABLE
u8SampleSelect	Sampling interval in terms of divided clock periods: E_AHI_AP_SAMPLE_2 (2 clock periods) E_AHI_AP_SAMPLE_4 (4 clock periods) E_AHI_AP_SAMPLE_6 (6 clock periods) E_AHI_AP_SAMPLE_8 (8 clock periods)
u8ClockDivRatio	Clock divisor (frequencies based on16MHz peripheral clock): E_AHI_AP_CLOCKDIV_2MHZ (divisor of 8) E_AHI_AP_CLOCKDIV_1MHZ (divisor of 16) E_AHI_AP_CLOCKDIV_500KHZ (divisor of 32) E_AHI_AP_CLOCKDIV_250KHZ (divisor of 64) (500kHz is recommended for ADC)

bRefSelect	Source of reference voltage, V_{ref}: E_AHI_AP_EXTREF (external from VREF pin) E_AHI_AP_INTREF (internal)

Returns

None

vAHI_ApSetBandGap

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

Description

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

Note that:

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

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

Parameters

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

Returns

None

bAHI_APRegulatorEnabled

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

Description

This function enquires whether the regulator used to power the analogue peripherals has powered up. The function should be called after enabling the regulator through **vAHI_ApConfigure()**. When the regulator is enabled, it will take a little time to start - this period is 16µs.

Parameters

None

Returns

TRUE if powered up, FALSE if still waiting

vAHI_ApSetSlowConversions

```
void vAHI_ApSetSlowConversions (
                    bool_t bSlowConversionsEnable);
```

Description

This function can be used to enable or disable slow conversions. When enabled, the conversions are slowed (i.e. take longer) by a factor of 8.

Parameters

bSlowConversionsEnable	Enable/disable slow conversions: TRUE - enable slow conversions FALSE - disable slow conversions

Returns

None

vAHI_APRegisterCallback

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

Description

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

> **Note:** Among the analogue peripherals, only the ADC generates Analogue Peripheral interrupts. The comparator generates System Controller interrupts (see Section 3.5).

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId*, **uint32** *u32ItemBitmap***);**

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

Interrupt handling and the callback function prototype are described in Appendix A. Analogue peripheral interrupt handling is further described in Section 4.4.

Parameters

prApCallback	Pointer to callback function to be registered

Returns

None

20.2 ADC Functions

This section describes the functions that can be used to control the on-chip 10-bit ADC (Analogue-to-Digital Converter). The ADC can be switched between 6 different sources - 4 pins on the device, an on-chip temperature sensor and a voltage monitor. The ADC can be configured to perform a single conversion or convert continuously (until stopped). It is also possible to operate the ADC in accumulation mode, in which a number of consecutive samples are added together for averaging.

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

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

Note 2: When an ADC input which is shared with a DIO is used, the associated DIO should be configured as an input with the pull-up disabled (using DIO functions detailed in Chapter 21).

vAHI_AdcEnable

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

Description

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

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

- **Single-shot mode:** ADC will perform a single conversion and then stop
- **Continuous mode:** ADC will perform conversions repeatedly until stopped using the function **vAHI_AdcDisable()**

If using the ADC in accumulation mode then the mode set here is ignored.

The function also allows the input source for the ADC to be selected as one of six pins on the JN517x device, the on-chip temperature sensor or the internal voltage monitor. The voltage range for the analogue input to the ADC can also be selected as 0 to V_{ref} or 0 to $2V_{ref}$.

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

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

Parameters

bContinuous	Conversion mode of ADC: E_AHI_ADC_CONTINUOUS (continuous mode) E_AHI_ADC_SINGLE_SHOT (single-shot mode)
bInputRange	Input voltage range: E_AHI_AP_INPUT_RANGE_1 (0 to V_{ref}) E_AHI_AP_INPUT_RANGE_2 (0 to $2V_{ref}$)
u8Source	Source for conversions: E_AHI_ADC_SRC_ADC5 (ADC5 input) E_AHI_ADC_SRC_ADC4 (ADC4 input) E_AHI_ADC_SRC_ADC3 (ADC3 input) E_AHI_ADC_SRC_ADC2 (ADC2 input) E_AHI_ADC_SRC_ADC1_VREF_EXT (ADC1 input - via VREF pin) E_AHI_ADC_SRC_ADC0_TEST1 (ADC0 input) E_AHI_ADC_SRC_TEMP (on-chip temperature sensor) E_AHI_ADC_SRC_VOLT (internal voltage monitor) E_AHI_ADC_SRC_TM_0 (test mode 1 - not used) E_AHI_ADC_SRC_TM_1 (test mode 2 - not used)

Returns

None

vAHI_AdcSetTemperatureSensor

```
void vAHI_ApSetTemperatureSensor(bool_t bEnable);
```

Description

This function can be used to enable or disable the temperature sensor independently of the function **vAHI_AdcEnable()**.

Parameters

bEnable	Enable/disable temperature sensor: TRUE - enable temperature sensor FALSE - disable temperature sensor

Returns

None

vAHI_AdcSetBatteryMonitor

void vAHI_ApSetBatteryMonitor(bool_t *bEnable***);**

Description

This function can be used to enable or disable the battery monitor independently of the function **vAHI_AdcEnable()**.

Parameters

bEnable	Enable/disable battery monitor: TRUE - enable battery monitor FALSE - disable battery monitor

Returns

None

vAHI_AdcStartSample

```
void vAHI_AdcStartSample(void);
```

Description

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

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

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

Note: If you wish to use the ADC in accumulation mode, start sampling using **vAHI_AdcStartAccumulateSamples()** instead.

Parameters

None

Returns

None

vAHI_AdcStartAccumulateSamples

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

Description

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

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

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

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

Parameters

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

Returns

None

bAHI_AdcPoll

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

Description

This function can be used when the ADC is operating in single-shot mode, continuous mode or accumulation mode, to check whether the ADC is still busy performing a conversion:

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

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

Parameters

None

Returns

TRUE if ADC is busy

FALSE if conversion complete

u16AHI_AdcRead

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

Description

This function reads the most recent ADC conversion result.

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

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

Parameters

None

Returns

Most recent single conversion result or accumulated conversion result:

- A single conversion result is contained in the least significant 10 bits of the 16-bit returned value
- An accumulated conversion result is contained in the least significant 14 bits of the 16-bit returned value

vAHI_AdcDisable

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

Description

This function disables the ADC. It can be used to stop the ADC when operating in continuous mode or accumulation mode.

Parameters

None

Returns

None

20.3 ADC with DMA Engine Functions

This section describes the functions that can be used to control the on-chip 10-bit ADC (Analogue-to-Digital Converter) when used in conjunction with the DMA engine, in 'sample buffer mode'. In this mode, ADC data samples are produced at regular intervals and transferred into a buffer in RAM as 16-bit samples, where this data transfer and storage is performed by the DMA engine independently of the CPU.

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

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

Note 2: When an ADC input which is shared with a DIO is used, the associated DIO should be configured as an input with the pull-up disabled (using DIO functions detailed in Chapter 21).

bAHI_AdcEnableSampleBuffer

```
bool_t bAHI_AdcEnableSampleBuffer(
                    bool_t bInputRange,
                    uint8 u8SourceBitmap,
                    uint16 *pu16Buffer,
                    uint16 u16BufferSize,
                    bool_t bBufferWrap,
                    uint8 u8InterruptModes);
```

Description

This function configures and starts the ADC in sample buffer mode, in which 10-bit samples are produced repeatedly by the ADC and are transferred into a RAM buffer by the DMA engine as 16-bit samples.

Sampling is triggered by a dedicated timer, which is known as the Analogue Peripheral Timer (APT) and has the designation Timer8.

The function allows the input source(s) for the ADC to be selected from a number of external input pins (shared with DIOs), the on-chip temperature sensor and the internal voltage monitor. Sample buffer mode allows multiple inputs to be selected (through a bitmap) and multiplexed - in this case, on each timer trigger, samples will be produced from each of the selected inputs, in turn, and written to the buffer. The inputs are sampled in the following order: ADC0 input through to ADC5 input, temperature sensor, voltage monitor. Note that the internal voltage monitor measures the voltage on the pin VDD1.

The RAM buffer must be specified in terms of a pointer to the start of the buffer and the size of the buffer (in 16-bit samples, up to a maximum of 2047). The option for buffer to wrap around can also be selected - in this case, once the buffer is full, data will be written to the start of the buffer again. If this option is not selected, conversions will stop once the buffer is full.

The condition(s) on which DMA interrupts will be generated can also be selected. These interrupts reflect the state of the RAM buffer and at least one must be selected:

- Buffer half-full
- Buffer full
- Buffer overflow (can be used when the buffer wrap option is disabled)

These interrupts must be serviced by the user-defined callback function registered using **vAHI_APRegisterCallback()**.

The voltage range for the analogue input to the ADC can also be selected as 0 to V_{ref} or 0 to $2V_{ref}$. Note that the source of V_{ref} is defined using **vAHI_ApConfigure()**.

Before starting the ADC using this function, the regulator used to power the analogue peripherals must have been enabled using the function **vAHI_ApConfigure()**. You must also check that the regulator has started, using the function **bAHI_APRegulatorEnabled()**.

Parameters

bInputRange	Input voltage range: E_AHI_AP_INPUT_RANGE_1 (0 to V_{ref}) E_AHI_AP_INPUT_RANGE_2 (0 to $2V_{ref}$)
u8SourceBitmap	Source(s) for conversions - any logical-ORed combination of: E_AHI_ADC_DMA_SRC_ADC5_MASK E_AHI_ADC_DMA_SRC_ADC4_MASK E_AHI_ADC_DMA_SRC_ADC3_MASK E_AHI_ADC_DMA_SRC_ADC2_MASK E_AHI_ADC_DMA_SRC_ADC1_VREF_EXT_MASK E_AHI_ADC_DMA_SRC_ADC0_TEST1_MASK E_AHI_ADC_DMA_SRC_TEMP_MASK E_AHI_ADC_DMA_SRC_VOLT_MASK
**pu16Buffer*	Pointer to start of RAM buffer in which samples will be stored
u16BufferSize	Size of RAM buffer, in 16-bit samples (valid range is 1-2047)
bBufferWrap	Indicates whether buffer will wrap around: TRUE - Buffer wrap enabled FALSE - Buffer wrap disabled
u8InterruptModes	DMA interrupt(s) to be generated (must select at least one): E_AHI_AP_INT_DMA_OVER_MASK (buffer overflow) E_AHI_AP_INT_DMA_END_MASK (buffer full) E_AHI_AP_INT_DMA_MID_MASK (buffer half-full)

Returns

TRUE - conversions successfully started

FALSE - conversions not successfully started (e.g. parameter error)

vAHI_AdcDisableSampleBuffer

```
void vAHI_AdcDisableSampleBuffer(void);
```

Description

This function can be used to stop the ADC operation when it has been started in sample buffer mode using the function **bAHI_AdcEnableSampleBuffer()**. In particular, the function can be used to stop the ADC when buffer wrap has been enabled and, therefore, the ADC will otherwise operate indefinitely.

Parameters

None

Returns

None

u16AHI_AdcSampleBufferOffset

```
uint16 u16AHI_AdcSampleBufferOffset(void);
```

Description

This function can be used in sample buffer mode to obtain the location in the RAM buffer where the next sample will be written. The function is primarily intended to be used as a diagnostic tool during application development to determine the progress of the DMA transfer to the buffer.

The location is returned as an offset (in 16-bit samples) from the start of the buffer. Note that when the buffer is full:

- if buffer wrap is not enabled, the returned value will be 2047
- if buffer wrap is enabled, the returned value will be 0

Parameters

None

Returns

Offset of next free location from start of buffer (range of possible values is 0 to 2047)

20.4 Comparator Functions

This section describes the functions that can be used to control the on-chip comparator.

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

> **Note 1:** If the comparator is to be used to wake the device from sleep mode then only the comparator '+' and '-' pins can be used. The internal reference voltage cannot be used.
>
> **Note 2:** The analogue peripherals regulator must be enabled while configuring a comparator, although it can be disabled once configuration is complete.
>
> **Note 3:** When a comparator pin is used, the associated DIO should be configured as an input with the pull-up disabled (using DIO functions detailed in Chapter 21).

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

vAHI_ComparatorEnable

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

Description

This function configures and enables the comparator. The input signal, reference signal and hysteresis setting must be specified.

The external input signal to be monitored can be provided on the comparator '+' pin (COMP1P) or '-' pin (COMP1M). This signal is compared with a reference signal which is either an external input on the other comparator pin or the internal reference voltage (V_{ref}). The input and reference signals are selected through a single parameter (*u8SignalSelect*) using one of following enumerations:

Input Signal	Reference Signal	Enumeration (*u8SignalSelect*)
COMP1P	COMP1M	E_AHI_COMP_SEL_EXT
COMP1P	V_{ref}	E_AHI_COMP_SEL_BANDGAP
COMP1M	COMP1P	E_AHI_COMP_SEL_EXT_INVERSE
COMP1M	V_{ref}	E_AHI_COMP_SEL_BANDGAP_INVERSE

The hysteresis voltage selected should be greater than:

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

Note: This function puts the comparator into standard-power mode in which it draws 73µA of current. The comparator can subsequently be put into low-power mode, in which it draws 0.8µA of current, by calling the function **vAHI_ComparatorLowPowerMode()**.

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

Parameters

u8Comparator	Identity of comparator: E_AHI_AP_COMPARATOR_1
u8Hysteresis	Hysteresis setting (controllable from 0 to 40mV) E_AHI_COMP_HYSTERESIS_0MV (0mV) E_AHI_COMP_HYSTERESIS_10MV (10mV) E_AHI_COMP_HYSTERESIS_20MV (20mV) E_AHI_COMP_HYSTERESIS_40MV (40mV)
u8SignalSelect	Selection of input and reference signals (see table above): E_AHI_COMP_SEL_EXT E_AHI_COMP_SEL_BANDGAP E_AHI_COMP_SEL_EXT_INVERSE E_AHI_COMP_SEL_BANDGAP_INVERSE

Returns

None

vAHI_ComparatorDisable

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

Description

This function disables the comparator.

Parameters

u8Comparator	Identity of comparator: E_AHI_AP_COMPARATOR_1

Returns

None

vAHI_ComparatorLowPowerMode

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

Description

This function can be used to enable or disable low-power mode on the comparator.

In low-power mode, a comparator draws 0.8µA of current, compared with 73µA when operating in standard-power mode. Low-power mode is ideal for energy harvesting. The mode is also automatically enabled when the device is sleeping.

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

Parameters

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

Returns

None

vAHI_ComparatorIntEnable

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

Description

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

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

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

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

Parameters

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

Returns

None

vAHI_ComparatorWakeEnable

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

Description

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

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

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

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

Parameters

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

Returns

None

u8AHI_ComparatorStatus

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

Description

This function obtains the status of the comparator.

To obtain the status of the comparator, the returned value must be bitwise ANDed with the mask E_AHI_AP_COMPARATOR_MASK_1.

The result is interpreted as follows:

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

Parameters

None

Returns

Value containing the status of comparator (see above)

u8AHI_ComparatorWakeStatus

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

Description

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

To obtain the wake-up interrupt status of the comparator, the returned value must be bitwise ANDed with the mask E_AHI_AP_COMPARATOR_MASK_1.

The result is interpreted as follows:

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

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

Parameters

None

Returns

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

bAHI_ComparatorControl

```
bool_t bAHI_ComparatorControl(
        bool_t bEnable,
        uint8 u8ComparatorSourceBitmap,
        uint32 u32ComparatorInterruptModes,
        uint16 u16DC_UT,
        uint16 u16DC_LT);
```

Description

This function is used to set up the comparator to show when a source analogue input matches the condition set up in the Comparator Interrupt Modes when compared to the Upper and Lower Threshold voltages.

The function also allows the comparator to be enabled or disabled. When enabled, the function sets up the source(s) for the comparator input hardware.

Parameters

bEnable	Enable/disable the comparator: TRUE - enable comparator FALSE - disable comparator
u8ComparatorSourceBitmap	Bitmap specifying source signals - a logical-ORed combination of: E_AHI_ADC_DMA_SRC_ADC5_MASK E_AHI_ADC_DMA_SRC_ADC4_MASK E_AHI_ADC_DMA_SRC_ADC3_MASK E_AHI_ADC_DMA_SRC_ADC2_MASK E_AHI_ADC_DMA_SRC_ADC1_VREF_EXT_MASK E_AHI_ADC_DMA_SRC_ADC0_TEST1_MASK E_AHI_ADC_DMA_SRC_TEMP_MASK E_AHI_ADC_DMA_SRC_VOLT_MASK
u32ComparatorInterruptModes	Bitmap specifying conditions that can trigger an interrupt - a logical-ORed combination of: E_AHI_AP_INT_DC_LOW_MASK - interrupt when selected ADC channel is below lower threshold E_AHI_AP_INT_DC_HIGH_MASK - interrupt when selected ADC channel is exceeds upper threshold E_AHI_AP_INT_DC_INRANGE_MASK - interrupt when selected ADC channel is between upper and lower thresholds E_AHI_AP_INT_DC_OFFRANGE_MASK - interrupt when selected ADC channel is outside upper and lower thresholds
u16DC_UT	10-bit value for the upper threshold of the digital comparator
u16DC_LT	10-bit value for the lower threshold of the digital comparator

Returns

TRUE if the comparator is successfully set up and enabled

FALSE if the comparator is disabled or parameter error

21. DIO and DO Functions

This chapter describes the functions that can be used to control the digital input/output lines, referred to as DIOs, and the digital output lines, referred to as DOs. The JN517x microcontroller has:

- 18 Digital Input/Outputs, numbered DIO0 to DIO15, DIO17 and DIO18
- 2 Digital Outputs, numbered DO0 and DO1

Each DIO/DO can be individually configured. However, the pins for the DIO/DO lines are shared with other peripherals (see Chapter 5) and are not available when those peripherals are enabled. For details of the shared pins, refer to the data sheet for your microcontroller. In addition to normal operation, when configured as inputs, the DIOs can be used to generate interrupts and wake the device from sleep.

> **Note:** For guidance on using the DIO functions in JN517x application code, refer to Chapter 5.

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

In some of the above functions, a 32-bit bitmap is used to represent the set of DIOs. In the bitmap, each of bits 0 to 15, 17 and 18 represents a DIO pin, where bit 0 represents DIO0 and bit 18 represents DIO18 (bits 16 and 19-31 are unused). The bitmap is shown in Table 10 below.

Bit	DIO
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14
15	15
16	Reserved
17	17
18	18
19-31	Reserved

Table 10: Bitmap Representation of DIOs

vAHI_DioSetDirection

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

Description

This function sets the direction for the DIO pins individually as either input or output (note that they are set as inputs, by default, on reset). This is done through two bitmaps for inputs and outputs, *u32Inputs* and *u32Outputs* respectively. In these values, each bit represents a DIO pin, as described on page 217. Setting a bit in one of these bitmaps configures the corresponding DIO as an input or output, depending on the bitmap.

Note that:

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

Parameters

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

Returns

None

vAHI_DioSetOutput

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

Description

This function sets individual DIO outputs on or off, driving an output high or low, respectively. This is done through two bitmaps for on-pins and off-pins, *u32On* and *u32Off* respectively. In these values, each bit represents a DIO pin, as described on page 217. Setting a bit in one of these bitmaps configures the corresponding DIO output as on or off, depending on the bitmap.

Note that:

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

Parameters

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

Returns

None

vAHI_DioSetPullup

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

Description

This function sets the pull-ups/pull-downs on individual DIO pins as on or off. A pull-up/pull-down can be set irrespective of whether the pin is an input or output. This is done through two bitmaps for ‘pull-ups/pull-downs on’ and ‘pull-ups/pull-downs off’, *u32On* and *u32Off* respectively. In these values, each bit represents a DIO pin, as indicated in Table 10 on page 218.

Note that:

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

Parameters

u32On	Bitmap of ‘pull-ups/pull-downs on’ - a bit set means that the corresponding pull-up/pull-down will be enabled
u32Off	Bitmap of ‘pull-ups/pull-downs off’ - a bit set means that the corresponding pull-up/pull-down will be disabled

Returns

None

vAHI_DioSetPullupDirection

void vAHI_DioSetPullupDirection(uint32 *u32Up***,**
uint32 *u32Down***);**

Description

This function selects the pull-up or pull-down function on individual DIO pins. A pull-up or pull-down can be set irrespective of whether the pin is an input or output. This is done through two bitmaps for selecting 'pull-up function enabled' and 'pull-down function enabled', *u32Up* and *u32Down* respectively. In these values, each bit represents a DIO pin, as indicated in Table 10 on page 218. Setting a bit in one of these bitmaps configures the corresponding DIO output as on or off, depending on the bitmap.

Note that:

- The DIOs have a mixture of pull-ups and pull-downs as their default states after system reset. DIOs 3, 7, 8, 12, 13 and 14 are pull-downs and the remainder are pull-ups. The pull-ups/pull-downs are enabled by a call to **vAHI_DioSetPullup()**. It is recommended that the pull-up direction is set prior to the pull-up/pull-down function being enabled, in order to ensure that the DIO is in the correct state to work as intended with external circuitry.
- Not all DIO pins must be defined (in other words, *u32Up* bitwise ORed with *u32Down* does not need to produce all ones for the DIO bits).
- Any DIO pins that are not defined by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Up* and *u32Down*, the DIO pin will use a pull-up.

Parameters

u32Up	Bitmap of pull-up pins - a bit set means that the corresponding DIO will have a pull-up selected
u32Down	Bitmap of pull-down pins - a bit set means that the corresponding DIO pin will have a pull-down selected

Returns

None

u32AHI_DioReadPullupDirection

```
uint32 u32AHI_DioReadPullupDirection(void);
```

Description

This function reads back the pull-up/pull-down function on the individual DIO pins and returns a bitmap, as described on Table 10 on page 218, representing the state of the register controlling the pull-ups and pull-downs.

A bit which is set to 1 indicates a pull-up has been selected, while a bit which is set to 0 indicates a pull-down.

In both cases, neither setting will take effect unless the pull-up/pull-down is set to active (see **vAHI_DioSetPullup()**). By default, from reset, all pull-up/pull-down settings are active.

Parameters

None

Returns

Bitmap containing settings for DIO pull-ups and pull-downs

Note that the bitmap returned by this function needs to be combined with that from **vAHI_DioSetPullup()** in order to determine if the pull-up/pull-down setting is active.

vAHI_SetDIOpinMultiplexValue

```
void vAHI_SetDIOpinMultiplexValue(
                        uint8 u8DIO,
                        uint8 u8MultiplexerValue);
```

Description

This function is used select the function of a particular DIO pin. The DIO pins have multiple functions, controlled by a multiplexer block which routes the appropriate input or output to a selection of possible DIO pins. The mapping of these pins is shown in the table on the next page.

Note that the range of values that *u8DIO* can take is restricted - specifically the DOx pins are controlled by the function **vAHI_SetDOpinMultiplexValue()**, and the value range for *u8DIO* is restricted to 0-15, 17 and 18. Also note that some combinations of DIO pin and multiplexer value are not permitted and are greyed-out in the table below.

Parameters

u8DIO	Number of DIO being set up: 0-15, 17 or 18
u8MultiplexerValue	Selector value for the function required

Returns

None

DIO Number	Mux Selector Value							
	0	1	2	3	4	5	6	7
0	GPIO(IO)	RFRX(O)	SPI_M_SEL0(O)	FLICK_CTRL	ADO(O)			
1	GPIO(IO)	RFTX(O)		PC0(I)	ADE(O)			
2	GPIO(IO)	RFRX(O)	TIM0_CAP(I)	I2C_SDA(IO)	VAUX_SENSE	UART1_RX		
3	GPIO(IO)	RFTX(O)	TIM0_OUT(O)	I2C_SCL(IO)	FLICK_CTRL(O)	UART1_TX	PWM4/IR	
4	GPIO(IO)	UART0_RX(I)	TIM0CK_GT(I)	I2C_SCL(IO)	ADO(O)			
5	GPIO(IO)	UART0_TX(O)	TIM0_CAP(I)	I2C_SDA(IO)	ADE(O)	PC1(I)		
DO0	RSVD	GPIO25(0)	SPI_M_SCK(O)	ADE(O)				
DO1	RSVD	GPIO26(0)	SPI_M_MISO(I)	ADO(O)		SPI_S_MISO(O)		
6	GPIO(IO)	TIM1_CAP(I)	SPI_M_SEL0(O)	TCK(I)	UART0_CTS(I)	SPI_S_SCK(I)		UART1_RX(I)
7	GPIO(IO)		SPI_M_MOSI(O)	TDI(I)	SPI_M_SEL2(O)	SPI_S_SEL(I)	COMP_OUT(O)	BackupClk
8	GPIO(IO)	PWM5(O)	TIM0_OUT(O)			TRACECLK		32kHz xtal
9	GPIO(IO)	UART0_TX(O)		TDO(O) / TRACESWV				
10	GPIO(IO)	UART0_RX(I)		TDI(I)				
11	GPIO(IO)	TIM1_OUT(O)	SPI_M_SCK(O)	TMS(I)	UART0_RTS(O)	SPI_S_MOSI(I)	UART1_TX(O)	TRACED0
12	GPIO(IO)	PWM1/VAUXO	UART0_TX(O)					TRACED3
13	GPIO(IO)	PWM2	UART0_RX(I)			PC0(I)		TRACED2
14	GPIO(IO)	PWM3			SPI_S_MOSI(I)	PC1(I)	COMP_OUT(O)	TRACED1
15	GPIO(IO)	PWM6	SPI_M_MOSI(O)	TDO(O) / TRACESWV	SPI_M_SEL1(O)	TIM0CK_GT	VAUX_SENSE	SPI_S_SEL(I)
17	GPIO(IO)	TIM1_CAP(I)	SPI_M_SEL0(O)	TCK(I)				SPI_S_MISO(O)
18	GPIO(IO)	TIM1_OUT(O)	SPI_M_MISO(I)	TMS(I)				SPI_S_SCK(I)

u32AHI_ReadDIOMultiplexValue

uint32 u32AHI_ReadDIOMultiplexValue(uint8 *u8DIO***);**

Description

This function reads back the DIO multiplexor value for a particular DIO pin specified in the *u8DIO* parameter. The function returns a 4-bit number (only bits 0-2 are valid, bit 3 is reserved) which can be used to look up the function in use on the DIO from the table above, following the description of **vAHI_SetDIOpinMultiplexValue()**.

Parameters

u8DIO	Number of DIO being read: 0-15, 17 or 18

Returns

Value of multiplexor for the specified DIO.

If the DIO number specified is out of range, the function returns -1.

u32AHI_DioReadInput

uint32 u32AHI_DioReadInput (void);

Description

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

Parameters

None

Returns

Bitmap representing set of DIOs, as described Table 10 on page 218 - a bit is set to 1 if the corresponding DIO pin is high or to 0 if the pin is low (unused bits are always 0).

vAHI_DioSetByte

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

Description

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

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

Parameters

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

Returns

None

u8AHI_DioReadByte

uint8 u8AHI_DioReadByte(bool_t *bDIOSelect*);

Description

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

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

Parameters

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

Returns

The byte read from DIO0-7 or DIO8-15

vAHI_DioInterruptEnable

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

Description

This function enables/disables interrupts on the DIO pins - that is, whether the signal on a DIO pin will generate an interrupt. This is done through two bitmaps for 'interrupts enabled' and 'interrupts disabled', *u32Enable* and *u32Disable* respectively. In these values, each bit represents a DIO pin, as described in Table 10 on page 218. Setting a bit in one of these bitmaps enables/disables interrupts on the corresponding DIO, depending on the bitmap (by default, all DIO interrupts are disabled).

Note that:

- Not all DIO interrupts must be defined (in other words, *u32Enable* bitwise ORed with *u32Disable* does not need to produce all ones for bits 0-18).
- Any DIO interrupts that are not defined by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Enable* and *u32Disable*, the corresponding DIO interrupt will default to disabled.
- This call has no effect on DIO pins that are not defined as inputs (see **vAHI_DioSetDirection()**).
- DIOs assigned to enabled JN517x peripherals are affected by this function.
- The DIO interrupt settings made with this function are retained during sleep.

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

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

***Caution:** This function has the same effect as **vAHI_DioWakeEnable()** - both functions access the same JN517x register bits. Therefore, do not allow the two functions to conflict in your code.*

Parameters

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

Returns

None

vAHI_DioInterruptEdge

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

Description

This function configures enabled DIO interrupts by controlling whether individual DIOs will generate interrupts on a rising or falling edge of the DIO signal. This is done through two bitmaps for 'rising edge' and 'falling edge', *u32Rising* and *u32Falling* respectively. In these values, each bit represents a DIO pin, as described in Table 10 on page 218. Setting a bit in one of these bitmaps configures interrupts on the corresponding DIO to occur on a rising or falling edge, depending on the bitmap (by default, all DIO interrupts are 'rising edge').

Note that:

- Not all DIO interrupts must be configured (in other words, *u32Rising* bitwise ORed with *u32Falling* does not need to produce all ones for the DIO bits).
- Any DIO interrupts that are not configured by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Rising* and *u32Falling*, the corresponding DIO interrupt will default to 'rising edge'.
- This call has no effect on DIO pins that are not defined as inputs (see **vAHI_DioSetDirection()**).
- DIOs assigned to enabled JN517x peripherals are affected by this function.
- The DIO interrupt settings made with this function are retained during sleep.

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

> ***Caution:*** *This function has the same effect as* ***vAHI_DioWakeEdge()*** *- both functions access the same JN517x register bits. Therefore, do not allow the two functions to conflict in your code.*

Parameters

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

Returns

None

u32AHI_DioInterruptStatus

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

Description

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

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

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

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

> **Note:** This function has the same effect as **vAHI_DioWakeStatus()** - both functions access the same JN517x register bits.

Parameters

None

Returns

Bitmap representing set of DIOs, as described in Table 10 on page 218 - a bit is set to 1 if the corresponding DIO interrupt has occurred or to 0 if the interrupt has not occurred (unused bits are always 0).

vAHI_DioWakeEnable

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

Description

This function enables/disables wake interrupts on the DIO pins - that is, whether activity on a DIO input will be able to wake the device from Sleep or Doze mode. This is done through two bitmaps for 'wake enabled' and 'wake disabled', *u32Enable* and *u32Disable* respectively. In these values, each bit represents a DIO pin, as described in Table 10 on page 218. Setting a bit in one of these bitmaps enables/disables wake interrupts on the corresponding DIO, depending on the bitmap.

Note that:

- Not all DIO wake interrupts must be defined (in other words, *u32Enable* bitwise ORed with *u32Disable* does not need to produce all ones for the DIO bits).
- Any DIO wake interrupts that are not defined by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Enable* and *u32Disable*, the corresponding DIO wake interrupt will default to disabled.
- This call has no effect on DIO pins that are not defined as inputs (see **vAHI_DioSetDirection()**).
- DIOs assigned to enabled JN517x peripherals are affected by this function.
- The DIO wake interrupt settings made with this function are retained during sleep.

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

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

Caution: *This function has the same effect as* ***vAHI_DioInterruptEnable()*** *- both functions access the same JN517x register bits. Therefore, do not allow the two functions to conflict in your code.*

Parameters

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

Returns

None

vAHI_DioWakeEdge

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

Description

This function configures enabled DIO wake interrupts by controlling whether individual DIOs will generate wake interrupts on a rising or falling edge of the DIO input. This is done through two bitmaps for ‘rising edge’ and ‘falling edge’, *u32Rising* and *u32Falling* respectively. In these values, each bit represents a DIO pin, as described in Table 10 on page 218. Setting a bit in one of these bitmaps configures wake interrupts on the corresponding DIO to occur on a rising or falling edge, depending on the bitmap (by default, all DIO wake interrupts are ‘rising edge’).

Note that:

- Not all DIO wake interrupts must be configured (in other words, *u32Rising* bitwise ORed with *u32Falling* does not need to produce all ones for the DIO bits).
- Any DIO wake interrupts that are not configured by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Rising* and *u32Falling*, the corresponding DIO wake interrupt will default to ‘rising edge’.
- This call has no effect on DIO pins that are not defined as inputs (see **vAHI_DioSetDirection()**).
- DIOs assigned to enabled JN517x peripherals are affected by this function.
- The DIO wake interrupt settings made with this function are retained during sleep.

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

> ***Caution:*** *This function has the same effect as* ***vAHI_DioInterruptEdge()*** *- both functions access the same JN517x register bits. Therefore, do not allow the two functions to conflict in your code.*

Parameters

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

Returns

None

u32AHI_DioWakeStatus

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

Description

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

Note 1: If you wish to use this function to check whether a DIO caused a wake-up event, you must call it before **u32AHI_Init()**. Alternatively, you can determine the wake source as part of your System Controller callback function.

Note 2: When waking from deep sleep, this function will not indicate a DIO wake source because the device will have completed a full reset. When waking from sleep, the function may indicate more than one wake source if multiple DIO events occurred while the device was booting.

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

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

Note: This function has the same effect as **vAHI_DioInterruptStatus()** - both functions access the same JN517x register bits.

Parameters

None

Returns

Bitmap representing set of DIOs, as described in Table 10 on page 218 - a bit is set to 1 if the corresponding DIO wake interrupt has occurred or to 0 if the interrupt has not occurred (unused bits are always 0).

vAHI_SetDIOpinsRadioMode

```
void vAHI_SetDIOpinsRadioMode(bool_t boEnable);
```

Description

This function can be used to enable or disable the mode which allows access to the radio and modem interface signals via the DIO pins.

Parameters

boEnable	Enable/disable mode: TRUE - enable FALSE - disable (set the interface back to normal operation)

Returns

None

bAHI_DoEnableOutputs

```
bool_t bAHI_DoEnableOutputs(bool_t bEnable);
```

Description

This function can be used to enable both digital output pins (DO0 and DO1) for general-purpose use.

When enabled for general-purpose use, these pins cannot be used by the SPI Master and Antenna Diversity switching.

Note: From reset, during sleep and on waking from sleep, the DO pins revert to being disabled as general-purpose outputs with pull-ups enabled.

Parameters

bEnable	Enable or disable digital outputs: TRUE - enable FALSE - disable

Returns

TRUE - DO(s) successfully enabled

FALSE - DO(s) used by SPI Master, so not available to be driven

vAHI_DoSetDataOut

```
void vAHI_DoSetDataOut(uint8 u8On, uint8 u8Off);
```

Description

This function sets individual digital outputs (DO0 and DO1) on or off, driving an output high or low, respectively. This is done through two bitmaps for on-pins and off-pins, *u8On* and *u8Off* respectively. In these values, bit 0 represents the DO0 pin and bit 1 represents the DO1 pin. Setting a bit in one of these bitmaps configures the corresponding digital output as on or off, depending on the bitmap.

Note that:

- By default, the digital outputs are high (on) at power-up.
- Both DO pins do not need to be defined (in other words, *u8On* bitwise ORed with *u8Off* does not need to produce all ones for the DO bits).
- A DO pin that is not defined by a call to this function (the relevant bit being cleared in both bitmaps) will be left in its previous state.
- If a bit is set in both *u8On* and *u8Off*, the DO pin will default to off.
- If a DO is assigned to another peripheral which is enabled, this function call will not affect the relevant DO, until a time when the relevant peripheral is disabled.

Before this function is called, the function **bAHI_DoEnableOutputs()** must have been called to enable the relevant DO(s) and must have returned TRUE.

> **Note:** From reset, during sleep and on waking from sleep, the DO pins revert to being disabled as general-purpose outputs with pull-ups enabled.

Parameters

u8On	Bitmap of on-pins (only bits 0 and 1 are relevant) - a bit set means that the corresponding DO pin will be set to on
u8Off	Bitmap of off-pins (only bits 0 and 1 are relevant) - a bit set means that the corresponding DO pin will be set to off

Returns

None

vAHI_DoSetPullup

```
void vAHI_DoSetPullup(uint8 u8On, uint8 u8Off);
```

Description

This function sets the pull-ups on individual DO pins (DO0 and DO1) as on or off. This is done through two bitmaps for ‘pull-ups on’ and ‘pull-ups off’, *u8On* and *u8Off* respectively. In these values, bit 0 represents the DO0 pull-up and bit 1 represents the DO1 pull-up. Setting a bit in one of these bitmaps configures the corresponding DO pull-up as on or off, depending on the bitmap.

Note that:

- By default, the pull-ups are enabled (on) at power-up.
- Both DO pull-ups do not need to be set (in other words, *u8On* bitwise ORed with *u8Off* does not need to produce all ones for the DIO bits).
- A DO pull-ups that is not set by a call to this function (the relevant bit being cleared in both bitmaps) will be left in its previous state.
- If a bit is set in both *u8On* and *u8Off*, the corresponding DIO pull-up will default to off.

Before this function is called, the function **bAHI_DoEnableOutputs()** must have been called to enable the relevant DO(s) and must have returned TRUE. In addition, the SPI Master should not be subsequently enabled.

> **Note:** From reset, during sleep and on waking from sleep, the DO pins revert to being disabled as general-purpose outputs with pull-ups enabled.

Parameters

u8On	Bitmap of ‘pull-ups on’ (only bits 0 and 1 are relevant) - a bit set means that the corresponding pull-up will be enabled
u8Off	Bitmap of ‘pull-ups off’ (only bits 0 and 1 are relevant) - a bit set means that the corresponding pull-up will be disabled

Returns

None

vAHI_SetDOpinMultiplexValue

```
void vAHI_SetDOpinMultiplexValue(
                        uint8 u8DO,
                        uint8 u8MultiplexerValue);
```

Description

This function is used to select the function of a DO pin. The two DO pins have multiple functions, controlled by a multiplexer block which routes the appropriate input or output to one of the DO pins. The mapping of these pins is shown in the table following the description of **vAHI_SetDIOpinMultiplexValue()**.

Note that the values which *u8DO* can take are restricted to 0 and 1.

Also note that some combinations of DO pin and multiplexer value are not permitted and are greyed-out in the table.

Parameters

u8DO	Number of DO pin being set up (0 or 1)
u8MultiplexerValue	Selector value for the required function

Returns

None

22. UART Functions

This chapter details the functions for controlling the 16550-compatible UARTs (Universal Asynchronous Receiver Transmitters). The JN517x microcontroller has two UARTs, denoted UART0 and UART1, which can be independently enabled.

- UART0 uses four pins (shared with the DIOs) for the following signals: Transmit Data (TxD) output, Receive Data (RxD) input, Request-To-Send (RTS) output and Clear-To-Send (CTS) input. This UART can be used in 4-wire mode (using all four signals), 2-wire mode (using only the TxD and RxD signals) or 1-wire mode (using only the TxD signal). 4-wire mode is used to implement flow control and is the default mode.
- UART1 uses two pins (shared with the DIOs) for the following signals: Transmit Data (TxD) output and Receive Data (RxD) input. This UART can be used in 2-wire mode (using both signals) or 1-wire mode (using only the TxD signal). 2-wire mode is the default mode.

Note: For information on the UARTs and guidance on using the UART functions in JN517x application code, refer to Chapter 6.

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

bAHI_UartEnable

```
bool_t bAHI_UartEnable(uint8 u8Uart,
                       uint8 *pu8TxBufAd,
                       uint16 u16TxBufLen,
                       uint8 *pu8RxBufAd,
                       uint16 u16RxBufLen);
```

Description

This function enables the specified UART and configures the FIFO Transmit and Receive buffers for the UART. It enables the UART for use in 2-wire mode (no flow control implemented) and routes the TxD and RxD signals through fixed DIOs (see table below). It must be the first UART function called.

Note 1: If you wish to use alternative DIOs for the UART signals, you must call **bAHI_UartEnableNoneDIO()** instead, which does no DIO configuration itself.

Note 2: The function **bAHI_UartEnableNoneDIO()** must be called if you wish to use UART0 in 4-wire mode (with flow control) or either UART in 1-wire mode.

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

The UARTs should be operated from a peripheral clock which runs at 16MHz (i.e. the system clock should be sourced from an external crystal oscillator). Therefore, this system clock must be set up before calling this function (for clock set-up, refer to Section 3.1).

The function specifies the size (in bytes) and location in RAM of the Transmit and Receive FIFOs. The size of each buffer can be set between 16 and 2047 bytes (inclusive). Valid size and pointer values for the buffers must always be set.

The UARTs use the following sets of DIO lines for the TxD and RxD signals:

UART Signal	DIOs for UART0	DIOs for UART1
TxD	DIO9	DIO3
RxD	DIO10	DIO2

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
**pu8TxBufAd*	Pointer to start of Transmit FIFO
u16TxBufLen	Size of Transmit FIFO, in range 16 to 2047 bytes
**pu8RxBufAd*	Pointer to start of Receive FIFO
u16RxBufLen	Size of Receive FIFO, in range 16 to 2047 bytes

Returns

TRUE if UART was successfully intialised, FALSE if UART was not successfully initialised (e.g. UART specified via *u8Uart* is invalid or *u16TxBufLen* is outside valid range)

bAHI_UartEnableNoneDIO

```
bool_t bAHI_UartEnableNoneDIO(uint8 u8Uart,
                              uint8 *pu8TxBufAd,
                              uint16 u16TxBufLen,
                              uint8 *pu8RxBufAd,
                              uint16 u16RxBufLen);
```

Description

This function enables the specified UART and configures the FIFO Transmit and Receive buffers for the UART. The UARTs can be used in the following modes, which depend on the number of UART signals used:

- UART0 may use all four signals (CTS, RTS, TxD, RxD), in which case it is said to operate in 4-wire mode in which flow control is implemented
- UART0 and UART1 may use just two signals (TxD and RxD), in which case they are said to operate in 2-wire mode (in which no flow control is implemented)
- UART0 and UART1 may alternatively use just one signal (TxD), in which case they are said to operate in 1-wire mode (transmit only)

By default, the function enables the UART for use in 2-wire mode but it does not configure the use of any DIOs for the UART signals, which must be done separately (see below).

Note 1: If you wish to use a UART in 2-wire mode with the default DIOs, you may call **bAHI_UartEnable()** instead, which does the DIO configuration itself.

Note 2: This function, **bAHI_UartEnableNoneDIO()**, must be called if you wish to use UART0 in 4-wire mode or either UART in 1-wire mode, since **bAHI_UartEnable()** does not offer these modes.

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

The UARTs should be operated from a peripheral clock which runs at 16MHz (i.e. the system clock should be sourced from an external crystal oscillator). Therefore, this system clock must be set up before calling this function (for clock set-up, refer to Section 3.1).

The function specifies the size (in bytes) and location in RAM of the Transmit and Receive FIFOs. The size of each buffer can be set between 16 and 2047 bytes (inclusive). A valid size and pointer value for the Transmit FIFO must always be set. If the Receive FIFO is not required (e.g. in 1-wire mode) then its pointer value should be set to NULL (its size will be ignored in this case).

The UARTs may use the following sets of DIO lines (standard and alternative sets):

UART Signal	DIOs for UART0		DIOs for UART1	
CTS	DIO6	-	-	-
RTS	DIO11	-	-	-
TxD	DIO9	DIO12	DIO3	DIO11
RxD	DIO10	DIO13	DIO2	DIO6

The DIO pins used to carry the UART signals can be selected using the function **vAHI_SetDIOpinMultiplexValue()**, which must be called separately for each DIO pin used.

For both UARTs, 2-wire mode is enabled by default when **bAHI_UartEnableNoneDIO()** is called, but:

- If you wish to use UART0 in 4-wire mode, you will need to call **vAHI_SetDIOPinMultiplexValue()** to route the TxD and RxD signals of the UART to the required DIO pins and you will also need to call **vAHI_UartSetAutoFlowCtrl()** to enable Automatic Flow Control.
- If you wish to use UART0 or UART1 in 1-wire mode, you will need to call **vAHI_SetDIOPinMultiplexValue()** to route the TxD signal of the UART to the required DIO pin.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
**pu8TxBufAd*	Pointer to start of Transmit FIFO
u16TxBufLen	Size of Transmit FIFO, in range 16 to 2047 bytes
**pu8RxBufAd*	Pointer to start of Receive FIFO (if this FIFO is not needed, set to NULL)
u16RxBufLen	Size of Receive FIFO, in range 16 to 2047 bytes (this parameter is ignored when *pu8RxBufAd* is set to NULL)

Returns

TRUE if UART was successfully intialised, FALSE if UART was not successfully initialised (e.g. UART specified via *u8Uart* is invalid, *pu8TxBufAd* is set to NULL or *u16TxBufLen* is outside valid range)

vAHI_UartDisable

void vAHI_UartDisable(uint8 *u8Uart***);**

Description

This function disables the specified UART by powering it down.

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

Parameters

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

Returns

None

vAHI_UartSetBaudRate

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

Description

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

The possible baud-rates are:

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

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

Parameters

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

Returns

None

vAHI_UartSetBaudDivisor

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

Description

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

The baud-rate produced is defined by:

baud-rate = 1000000/*u16Divisor*

For example:

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

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

Parameters

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

u16Divisor Integer divisor

Returns

None

vAHI_UartSetClocksPerBit

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

Description

This function sets the baud-rate used by the specified UART to a value derived from a 16MHz peripheral clock. The function allows higher baud-rates to be set than those available through **vAHI_UartSetBaudRate()** and **vAHI_UartSetBaudDivisor()**.

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

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

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

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

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

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

Parameters

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

Returns

None

vAHI_UartSetClockDivisor

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

Description

This function sets the clock divisor for a UART in order to configure the baud rate. The baud rate is encoded as a set of pre-defined enumerations in **AppHardwareApi.h**.

Note that the values that *u8Uart* can take is restricted are restricted to 0 and 1. The valid range of *u8BaudRate* is also restricted

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
u16Divisor	Value selected from the following: E_AHI_UART_RATE_4800 E_AHI_UART_RATE_9600 E_AHI_UART_RATE_19200 E_AHI_UART_RATE_38400 E_AHI_UART_RATE_76800 E_AHI_UART_RATE_115200

Returns

TRUE if successful, FALSE otherwise

vAHI_UartSetControl

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

Description

This function sets various control bits for the specified UART.

Note that RTS for UART0 cannot be controlled automatically - it can only be set/cleared under software control (this setting will be ignored for UART1).

Parameters

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

Returns

None

vAHI_UartSetInterrupt

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

Description

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

Parameters

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

Returns

None

vAHI_UartSetRTSCTS

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

Description

This function instructs UART0 to take or release control of the DIO lines used for RTS and CTS in flow control (depending on the DIOs selected):

- DIO6 for CTS
- DIO11 for RTS

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

UART0 operates by default in 2-wire mode. If you wish to use this UART in 4-wire mode, it will be necessary to call **vAHI_UartSetRTSCTS()** before calling **bAHI_UartEnableNoneDIO()** in order to enable the RTS and CTS lines.

Parameters

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

Returns

None

vAHI_UartSetRTS

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

Description

This function instructs UART0 to set or clear its RTS signal in 4-wire mode.

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

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

Parameters

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

Returns

None

vAHI_UartSetAutoFlowCtrl

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

Description

This function allows Automatic Flow Control (AFC) to be configured and enabled for UART0 operating in 4-wire mode. The function parameters allow the following to be selected/set:

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

In order to use the RTS and CTS lines, UART0 must be enabled in 4-wire mode.

Parameters

u8Uart	Identity of UART: set to E_AHI_UART_0
u8RxFifoLevel	Receive FIFO automatic RTS trigger level: E_AHI_UART_FIFO_ARTS_LEVEL_8: 8 bytes E_AHI_UART_FIFO_ARTS_LEVEL_11: 11 bytes E_AHI_UART_FIFO_ARTS_LEVEL_13: 13 bytes E_AHI_UART_FIFO_ARTS_LEVEL_15: 15 bytes
bFlowCtrlPolarity	Active level (low or high) of RTS and CTS flow control: FALSE: RTS and CTS are active-low TRUE: RTS and CTS are active-high
bAutoRts	Enable/disable Automatic RTS feature: TRUE to enable FALSE to disable
bAutoCts	Enable/disable Automatic CTS feature: TRUE to enable FALSE to disable

Returns

None

vAHI_UartSetBreak

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

Description

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

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

Parameters

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

Returns

None

vAHI_UartReset

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

Description

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

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

Parameters

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

Returns

None

u16AHI_UartReadRxFifoLevel

uint16 u16AHI_UartReadRxFifoLevel(uint8 *u8Uart*);

Description

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

Parameters

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

Returns

Number of characters in the specified Receive FIFO

u16AHI_UartReadTxFifoLevel

uint16 u16AHI_UartReadTxFifoLevel(uint8 *u8Uart*);

Description

This function obtains the fill-level of the Transmit FIFO - that is, the number of characters currently in the FIFO.

Parameters

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

Returns

Number of characters in the specified Transmit FIFO

u8AHI_UartReadLineStatus

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

Description

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

Note that the following bits are cleared after reading:

E_AHI_UART_LS_ERROR
E_AHI_UART_LS_BI
E_AHI_UART_LS_FE
E_AHI_UART_LS_PE
E_AHI_UART_LS_OE

Parameters

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

Returns

Bitmap:

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

u8AHI_UartReadModemStatus

uint8 u8AHI_UartReadModemStatus(uint8 *u8Uart*);

Description

This function obtains modem status information from UART0 as a bitmap which includes the CTS and ‘CTS has changed’ status (which can be extracted as described below).

Parameters

u8Uart Identity of UART: set to E_AHI_UART_0

Returns

Bitmap in which:

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

u8AHI_UartReadInterruptStatus

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

Description

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

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

Parameters

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

Returns

Bitmap:

Bit range	Value/Enumeration	Description
Bit 0	0	More interrupts pending
	1	No more interrupts pending
Bits 1-3	E_AHI_UART_INT_RXLINE (3)	Receive line status interrupt (highest prioritry)
	E_AHI_UART_INT_RXDATA (2)	Receive data available interrupt (next highest priority)
	E_AHI_UART_INT_TIMEOUT (6)	Timeout interrupt (next highest priority)
	E_AHI_UART_INT_TX (1)	Transmit FIFO empty interrupt (next highest priority)
	E_AHI_UART_INT_MODEM (0)	Modem status interrupt (lowest priority)

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

vAHI_UartWriteData

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

Description

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

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

For UART0, if automatic flow control has been enabled for the local CTS line using the function **vAHI_UartSetAutoFlowCtrl()**, the data in the Transmit FIFO will only be transmitted once the CTS line has been asserted. In this case, **vAHI_UartWriteData()** can be called at any time to load data into the Transmit FIFO, provided that there is enough free space in the FIFO.

Refer to the description of **u16AHI_UartReadTxFifoLevel()** or **u8AHI_UartReadLineStatus()** for details of how to determine whether the Transmit FIFO already contains data.

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

Parameters

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

Returns

None

u8AHI_UartReadData

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

Description

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

Refer to the description of **u16AHI_UartReadRxFifoLevel()** or **u8AHI_UartReadLineStatus()** for details of how to determine whether the Receive FIFO is empty.

Parameters

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

Returns

Received byte

u16AHI_UartBlockWriteData

```
uint16 u16AHI_UartBlockWriteData(uint8 u8Uart,
                                 uint8 *pu8Data,
                                 uint16 u16DataLength);
```

Description

This function writes a block of data to the Transmit FIFO of the specified UART. The transmission of the data will then be handled by the on-chip DMA engine.

If no flow control or manual flow control is being implemented for data transmission, the data in the Transmit FIFO will be transmitted as soon as possible (irrespective of the state of the local CTS line). Therefore, **u16AHI_UartBlockWriteData()** should be called only when the destination device is able to receive the data.

For UART0, if automatic flow control has been enabled for the local CTS line using the function **vAHI_UartSetAutoFlowCtrl()**, the data in the Transmit FIFO will only be transmitted once the CTS line has been asserted. In this case, **u16AHI_UartBlockWriteData()** can be called at any time to load data into the Transmit FIFO, provided that there is enough free space in the FIFO.

Refer to the description of **u16AHI_UartReadTxFifoLevel()** or **u8AHI_UartReadLineStatus()** for details of how to determine whether the Transmit FIFO already contains data.

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

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
**pu8Data*	Pointer to start of data block to be written to Transmit FIFO
u16DataLength	Size of data block, in bytes

Returns

Number of bytes of data successfully written to the Transmit FIFO

u16AHI_UartBlockReadData

```
uint16 u16AHI_UartBlockReadData(
                        uint8 u8Uart,
                        uint8 *pu8DataBuffer,
                        uint16 u16DataBufferLength);
```

Description

This function reads a block of data from the Receive FIFO of the specified UART. If the FIFO is empty, the returned value is not valid.

A data buffer in RAM to receive the read data block must be specified.

Refer to the description of **u16AHI_UartReadRxFifoLevel()** or **u8AHI_UartReadLineStatus()** for details of how to determine whether the Receive FIFO is empty.

Parameters

u8Uart	Identity of UART: E_AHI_UART_0 (UART0) E_AHI_UART_1 (UART1)
**pu8DataBuffer*	Pointer to data buffer in RAM to receive read data block
u16DataBufferLength	Size of data buffer, in bytes

Returns

Number of bytes of data successfully read from Receive FIFO

vAHI_Uart0RegisterCallback

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

Description

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

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId*, **uint32** *u32ItemBitmap***);**

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

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

prUart0Callback	Pointer to callback function to be registered

Returns

None

vAHI_Uart1RegisterCallback

void vAHI_Uart1RegisterCallback(
PR_HWINT_APPCALLBACK *prUart1Callback***);**

Description

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

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId***, uint32** *u32ItemBitmap***);**

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

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

prUart1Callback	Pointer to callback function to be registered

Returns

None

23. Timer Functions

This chapter describes the functions that can be used to control the on-chip timers. The JN517x device has nine timers: Timer 0, Timer 1, PWM Timer 1, PWM Timer 2, PWM Timer 3, PWM Timer 4, PWM Timer 5, PWM Timer 6 and Analogue Peripheral Timer (APT). PWM Timers 1-6 have no external inputs and only support modes without inputs. The Analogue Peripheral Timer is dedicated to the Analogue Peripherals block. PWM Timers 1-6 are also known as Timers 2-7 and the Analogue Peripherals Timer is also known as Timer 8.

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

Note: For information on the timers and guidance on using the timer functions in JN517x application code, refer to Chapter 7.

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

vAHI_TimerEnable

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

Description

This function configures and enables the specified timer, and must be the first timer function called. The timer is derived from the peripheral clock, which can be divided down to produce the timer clock (a system clock sourced from the external crystal oscillator gives the most stable results). The timer can be used in various modes, introduced in Section 7.1 (note that some timers have no external inputs and therefore only support modes without inputs).

> **Note:** For PWM Timers 1-3, this function configures the relevant DIO for timer output (see below). Alternatively, you can call **vAHI_TimerEnableNoneDIO()**, which does no DIO configuration itself. For the other timers, the two enable functions have identical functionality.

The parameters of this enable function cover the following features:

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

  Timer clock frequency = Source clock frequency / $2^{u8Prescale}$

- **Interrupts** (*bIntRiseEnable* and *bIntPeriodEnable*): Interrupts can be generated:
  - in Timer or PWM mode, on a low-to-high transition (rising output) and/or on a high-to-low transition (end of the timer period)
  - in Counter mode, on reaching target counts

  You can register a user-defined callback function for timer interrupts using the function:

  vAHI_Timer0RegisterCallback() for Timer 0

  vAHI_Timer1RegisterCallback() for Timer 1

  vAHI_Timer2RegisterCallback() for Timer 2 (PWM1)

  vAHI_Timer3RegisterCallback() for Timer 3 (PWM2)

  vAHI_Timer4RegisterCallback() for Timer 4 (PWM3)

  vAHI_Timer5RegisterCallback() for Timer 5 (PWM4)

  vAHI_Timer6RegisterCallback() for Timer 6 (PWM5)

  vAHI_Timer7RegisterCallback() for Timer 7 (PWM6)

  vAHI_Timer8RegisterCallback() for Timer 8 (APT)

  Alternatively, timer interrupts can be disabled.

- **Timer output** (*bOutputEnable*): When operating in PWM mode or Delta-Sigma mode, the timer's signal is output on a DIO pin (see the table below), which must be enabled. If this option is enabled, the other DIOs associated with the timer cannot be used for general-purpose input/output.

Timer	Output DIO
PWM1	DIO12
PWM2	DIO13
PWM3	DIO14

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (PWM1) or E_PWM_TIMER1 E_AHI_TIMER_3 (PWM2) or E_PWM_TIMER2 E_AHI_TIMER_4 (PWM3) or E_PWM_TIMER3 E_AHI_TIMER_5 (PWM4) or E_PWM_TIMER4 E_AHI_TIMER_6 (PWM5) or E_PWM_TIMER5 E_AHI_TIMER_7 (PWM6) or E_PWM_TIMER6 E_AHI_TIMER_8 (APT)
u8Prescale	Prescale index, in range 0 to 16, used in dividing down source clock (divisor is $2^{u8Prescale}$)
bIntRiseEnable	Enable/disable interrupt on rising output (low-to-high): TRUE to enable FALSE to disable
bIntPeriodEnable	Enable/disable interrupt at end of timer period (high-to-low): TRUE to enable FALSE to disable
bOutputEnable	Enable/disable output of timer signal on DIO: TRUE to enable (PWM or Delta-Sigma mode) FALSE to disable (Timer mode)

Returns

None

vAHI_TimerEnableNoneDIO

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

Description

This function configures and enables the specified timer, and must be the first timer function called. The timer is derived from the peripheral clock, which can be divided down to produce the timer clock (a system clock sourced from the external crystal oscillator gives the most stable results). The timer can be used in various modes, introduced in Section 7.1 (note that some timers have no external inputs and therefore only support modes without inputs).

> **Note:** This function does not configure a DIO for the timer output and allows the DIO to be selected separately (see below). For PWM Timer 1-3, the alternative function **vAHI_TimerEnable()** does the DIO configuration itself. For the other timers, the two enable functions have identical functionality.

The parameters of this enable function cover the following features:

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

  Timer clock frequency = Source clock frequency / $2^{u8Prescale}$

- **Interrupts** (*bIntRiseEnable* and *bIntPeriodEnable*): Interrupts can be generated:
  - in Timer or PWM mode, on a low-to-high transition (rising output) and/or on a high-to-low transition (end of the timer period)
  - in Counter mode, on reaching target counts

  You can register a user-defined callback function for timer interrupts using the function:

  vAHI_Timer0RegisterCallback() for Timer 0

  vAHI_Timer1RegisterCallback() for Timer 1

  vAHI_Timer2RegisterCallback() for Timer 2 (PWM1)

  vAHI_Timer3RegisterCallback() for Timer 3 (PWM2)

  vAHI_Timer4RegisterCallback() for Timer 4 (PWM3)

  vAHI_Timer5RegisterCallback() for Timer 5 (PWM4)

  vAHI_Timer6RegisterCallback() for Timer 6 (PWM5)

  vAHI_Timer7RegisterCallback() for Timer 7 (PWM6)

  vAHI_Timer8RegisterCallback() for Timer 8 (APT)

  Alternatively, timer interrupts can be disabled.

- **Timer output** (*bOutputEnable*): When operating in PWM mode or Delta-Sigma mode, the timer's signal is output on a DIO pin (see Section 7.2.1), which must be enabled. If this option is enabled, the other DIOs associated with the timer cannot be used for general-purpose input/output.

Before the selected timer can be operated, the signals that the timer uses and generates must be routed to the IO pins selected to be used for the function. This is done by calling the function **vAHI_SetDIOpinMultiplexValue()**, described in Chapter 21.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (PWM1) or E_PWM_TIMER1 E_AHI_TIMER_3 (PWM2) or E_PWM_TIMER2 E_AHI_TIMER_4 (PWM3) or E_PWM_TIMER3 E_AHI_TIMER_5 (PWM4) or E_PWM_TIMER4 E_AHI_TIMER_6 (PWM5) or E_PWM_TIMER5 E_AHI_TIMER_7 (PWM6) or E_PWM_TIMER6 E_AHI_TIMER_8 (APT)
u8Prescale	Prescale index, in range 0 to 16, used in dividing down source clock (divisor is $2^{u8Prescale}$)
bIntRiseEnable	Enable/disable interrupt on rising output (low-to-high): TRUE to enable FALSE to disable
bIntPeriodEnable	Enable/disable interrupt at end of timer period (high-to-low): TRUE to enable FALSE to disable
bOutputEnable	Enable/disable output of timer signal on DIO: TRUE to enable (PWM or Delta-Sigma mode) FALSE to disable (Timer mode)

Returns

None

vAHI_TimerClockSelect

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

Description

This function can be used to enable/disable an external clock input for Timer 0. If enabled, the external input is taken from the DIO8 pin.

Note the following:

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

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

Parameters

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

Returns

None

vAHI_TimerConfigureOutputs

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

Description

This function configures certain parameters relating to the operation of the specified timer in the following modes (introduced in Section 7.1):

- **Timer mode:** The internal peripheral clock drives the timer's counter in order to produce a pulse cycle in either 'single shot' or 'repeat' mode. The clock may be temporarily interrupted by a gating input on a DIO (see Section 7.2.1 for the relevant DIOs). Clock gating can be enabled/disabled using this function for Timer 0 only (there are no gating inputs for Timers 1-7).
- **Pulse Width Modulation (PWM) mode:** The PWM signal produced in Timer mode (see above) is output, where this output can be enabled in **vAHI_TimerEnable()**. The signal is output on a DIO which depends on the timer selected (see Section 7.2.1 for the relevant DIOs). If required, the output signal can be inverted using this function on any of the timers operating in PWM mode.

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

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (PWM1) E_AHI_TIMER_3 (PWM2) E_AHI_TIMER_4 (PWM3) E_AHI_TIMER_5 (PWM4) E_AHI_TIMER_6 (PWM5) E_AHI_TIMER_7 (PWM6) E_AHI_TIMER_8 (APT)
bInvertPwmOutput	Enable/disable inversion of PWM output: TRUE to enable inversion FALSE to disable inversion
bGateDisable	Enable/disable external gating input for Timer mode: TRUE to disable clock gating input FALSE to enable clock gating input (for Timers 1-8, set to TRUE)

Returns

None

vAHI_TimerConfigureInputs

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

Description

This function configures certain parameters relating to the operation of Timer 0 (there are no external signal inputs for Timers 1-8) in the following modes (introduced in Section 7.1):

- **Capture mode:** An external signal is sampled on every tick of the timer. The results of the capture allow the period and pulse width of the sampled signal to be obtained. The input signal can be inverted using this function, allowing the low-pulse width to be measured (instead of the high-pulse width). This external signal is input as signal TIM0_CAP.
- **Counter mode:** The timer is used to count the number of transitions on an external input (selected using **vAHI_TimerClockSelect()**). This configure function allows selection of the transitions on which the count will be performed - on low-to-high transitions, or on both low-to-high and high-to-low transitions.

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

Parameters

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

Returns

None

vAHI_TimerStartSingleShot

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

Description

This function starts the specified timer in ‘single-shot’ mode. The function relates to Timer mode, PWM mode and Counter mode (introduced in Section 7.1).

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

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

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

In **Counter mode**, this function is used differently:

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

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

Note that Counter mode is only available for Timer 0.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (PWM1) E_AHI_TIMER_3 (PWM2) E_AHI_TIMER_4 (PWM3) E_AHI_TIMER_5 (PWM4) E_AHI_TIMER_6 (PWM5) E_AHI_TIMER_7 (PWM6) E_AHI_TIMER_8 (APT)
u16Hi	Number of clock periods after starting a timer before the output goes high (Timer or PWM mode) or count at which first interrupt generated (Counter mode)
u16Lo	Number of clock periods after starting a timer before the output goes low again (Timer or PWM mode) or count at which second interrupt generated and timer stops (Counter mode)

Returns

None

vAHI_TimerStartRepeat

```
void vAHI_TimerStartRepeat(uint8 u8Timer,
                           uint16 u32Hi,
                           uint16 u32Lo);
```

Description

This function starts the specified timer in 'repeat' mode. The function relates to Timer mode, PWM mode and Counter mode (introduced in Section 7.1).

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

1. The output is low until *u32Hi* clock periods have passed, when it goes high.
2. The output remains high until *u32Lo* clock periods have passed since the timer was started and then goes low again.

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

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

In **Counter mode**, this function is used differently:

- At a count of *u32Hi*, an interrupt (E_AHI_TIMER_RISE_MASK) will be generated (if enabled).
- At a count of *u32Lo*, another interrupt (E_AHI_TIMER_PERIOD_MASK) will be generated (if enabled) and the count will then be re-started from zero.

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

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

Note that Counter mode is only available for Timer 0.

When the timer is operating using the 16MHz clock the valid range for *u32Hi* and *u32Lo* is 16-bits; when operating with the 32MHz clock (enabled by **bAHI_Switch32MHzClockForPWM()**) the valid range for *u32Hi* and *u32Lo* is 17-bits.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (PWM1) E_AHI_TIMER_3 (PWM2) E_AHI_TIMER_4 (PWM3) E_AHI_TIMER_5 (PWM4) E_AHI_TIMER_6 (PWM5) E_AHI_TIMER_7 (PWM6) E_AHI_TIMER_8 (APT)
u32Hi	Number of clock periods after starting a timer before the output goes high (Timer or PWM mode) or count at which first interrupt generated (Counter mode)

u32Lo	Number of clock periods after starting a timer before the output goes low again (Timer or PWM mode) or count at which second interrupt generated (Counter mode)

Returns

None

vAHI_TimerStartCapture

void vAHI_TimerStartCapture(uint8 *u8Timer*);

Description

This function starts Timer 0 in Capture mode (Timers 1-8 cannot operate in Capture mode). This mode must first be configured using the function **vAHI_TimerConfigureInputs()**.

An input signal must be provided on the DIO9 pin. The incoming signal is timed and the captured measurements are:

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

These values are placed in registers to be read later using the function **vAHI_TimerReadCapture()** or **vAHI_TimerReadCaptureFreeRunning()**. They allow the input pulse width to be determined. If the timer is running with the 32 MHz clock enabled through a call to b**AHI_Switch32MHzClockForPWM()** then the 17-bit versions of these functions (**vAHI_Timer17bitReadCapture()** or **vAHI_Timer17bitReadCaptureFreeRunning()**) should be used.

Parameters

u8Timer	Identity of timer: set to E_AHI_TIMER_0

Returns

None

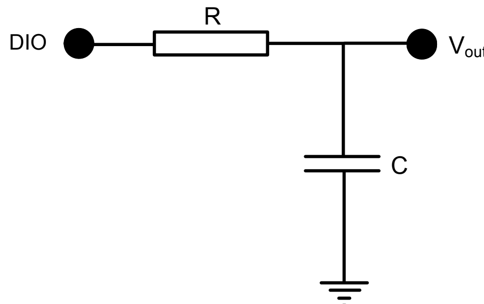
vAHI_TimerStartDeltaSigma

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

Description

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

To use this mode, the DIO output for the timer (see Section 7.2.1 for the relevant DIOs) must be enabled through **vAHI_TimerEnableNoneDIO()**. In addition, an RC circuit must be inserted on the DIO output pin in the arrangement shown below (also see Note below).



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

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

Note: For more information on 'Delta-Sigma' mode, refer to the JN517x Data Sheet.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (PWM1) E_AHI_TIMER_3 (PWM2) E_AHI_TIMER_4 (PWM3) E_AHI_TIMER_5 (PWM4) E_AHI_TIMER_6 (PWM5) E_AHI_TIMER_7 (PWM6) E_AHI_TIMER_8 (APT)
u16Hi	Number of 16MHz clock cycles for which the output will be high during a conversion period, in the range 0 to 65535 (full period is 65536 clock cycles)
0x0000	Fixed null value
bRtzEnable	Enable/disable RTZ (Return-to-Zero) option: TRUE to enable FALSE to disable

Returns

None

u16AHI_TimerReadCount

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

Description

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

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (PWM1) E_AHI_TIMER_3 (PWM2) E_AHI_TIMER_4 (PWM3) E_AHI_TIMER_5 (PWM4) E_AHI_TIMER_6 (PWM5) E_AHI_TIMER_7 (PWM6) E_AHI_TIMER_8 (APT)

Returns

Current count value of timer

vAHI_TimerReadCapture

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

Description

This function stops Timer 0 and then obtains the results from a 'capture' started using the function **vAHI_TimerStartCapture()**.

The values returned are offsets from the start of capture, as follows:

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

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

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

Capture mode and this function are relevant to Timer 0 and Timer 1 only.

Parameters

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

Returns

None

vAHI_TimerReadCaptureFreeRunning

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

Description

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

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

The values returned are offsets from the start of capture, as follows:

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

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

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

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

Capture mode and this function are relevant to Timer 0 and Timer 1 only.

Parameters

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

Returns

None

u32AHI_Timer17bitReadCount

uint32 u32AHI_Timer17bitReadCount(uint8 *u8Timer***);**

Description

This function performs a 17-bit read to obtain the current count value of the specified timer when running with the 32MHz clock, enabled by **bAHI_Switch32MHzClockForPWM()**.

Note that a 17-bit read is not available for Timer 5 (PWM4).

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (PWM1) E_AHI_TIMER_3 (PWM2) E_AHI_TIMER_4 (PWM3) E_AHI_TIMER_6 (PWM5) E_AHI_TIMER_7 (PWM6) E_AHI_TIMER_8 (APT)

Returns

Current count value of timer

vAHI_Timer17bitReadCapture

```
void vAHI_Timer17bitReadCapture(uint8 u8Timer,
                                uint16 *pu16Hi,
                                uint16 *pu16Lo);
```

Description

This function stops the selected timer and then performs a 17-bit read to obtain the results from a 'capture' started using the function **vAHI_TimerStartCapture()**. The function is used when running with the 32MHz clock, enabled by **bAHI_Switch32MHzClockForPWM()**.

The values returned are offsets from the start of capture, as follows:

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

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

If you wish to measure the pulse period of the input signal, you should use the function **vAHI_Timer17bitReadCaptureFreeRunning()**, which does not stop the timer.

Capture mode and this function are relevant to Timer 0 and Timer 1 only.

Parameters

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

Returns

None

vAHI_Timer17bitReadCaptureFreeRunning

```
void vAHI_Timer17bitReadCaptureFreeRunning(
                                    uint8 u8Timer,
                                    uint16 *pu16Hi,
                                    uint16 *pu16Lo);
```

Description

This function performs a 17-bit read to obtain the results from a 'capture' started on Timer 0 or 1 using the function **vAHI_TimerStartCapture().** The function does not stop the timer. This function is used when the timer is running with the 32MHz clock, enabled by **bAHI_Switch32MHzClockForPWM()**.

Alternatively, the function **vAHI_Timer17bitReadCapture()** can be used, which stops the timer before reporting the capture measurements.

The values returned are offsets from the start of capture, as follows:

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

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

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

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

Capture mode and this function are relevant to Timer 0 and Timer 1 only.

Parameters

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

Returns

None

bAHI_Switch32MHzClockForPWM

```
bool_t bAHI_Switch32MHzClockForPWM(
                        uint8 u8Timer,
                        bool_t bSwitchTo32MHz);
```

Description

This function allows the source clock to the nominated timer to be selected between 32MHz and 16MHz.

Timer 5 is excluded from this function due to the Infra-Red support on PWM4.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (PWM1) E_AHI_TIMER_3 (PWM2) E_AHI_TIMER_4 (PWM3) E_AHI_TIMER_6 (PWM5) E_AHI_TIMER_7 (PWM6) E_AHI_TIMER_8 (APT)
bSwitchTo32MHz	The source clock to switch to: TRUE - use 32 MHz clock FALSE - use 16 MHz clock

Returns

TRUE if successful, FALSE if *u8Timer* is out of range

vAHI_TimerStop

void vAHI_TimerStop (uint8 *u8Timer***);**

Description

This function stops the specified timer.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (PWM1) E_AHI_TIMER_3 (PWM2) E_AHI_TIMER_4 (PWM3) E_AHI_TIMER_5 (PWM4) E_AHI_TIMER_6 (PWM5) E_AHI_TIMER_7 (PWM6) E_AHI_TIMER_8 (APT)

Returns

None

vAHI_TimerDisable

```
void vAHI_TimerDisable (uint8 u8Timer);
```

Description

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

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

Parameters

u8Timer Identity of timer:
E_AHI_TIMER_0 (Timer 0)
E_AHI_TIMER_1 (Timer 1)
E_AHI_TIMER_2 (PWM1)
E_AHI_TIMER_3 (PWM2)
E_AHI_TIMER_4 (PWM3)
E_AHI_TIMER_5 (PWM4)
E_AHI_TIMER_6 (PWM5)
E_AHI_TIMER_7 (PWM6)
E_AHI_TIMER_8 (APT)

Returns

None

u8AHI_TimerFired

uint8 u8AHI_TimerFired(uint8 *u8Timer***);**

Description

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

> ***Caution:*** *This function should not be called within a Timer callback function which is invoked as the result of a Timer event, since the interrupt status of the timer is cleared before entering the callback function. The function should only be used when polling for the interrupt status of a timer.*

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (PWM1) E_AHI_TIMER_3 (PWM2) E_AHI_TIMER_4 (PWM3) E_AHI_TIMER_5 (PWM4) E_AHI_TIMER_6 (PWM5) E_AHI_TIMER_7 (PWM6) E_AHI_TIMER_8 (APT)

Returns

Bitmap:

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

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

vAHI_Timer0RegisterCallback

**void vAHI_Timer0RegisterCallback(
PR_HWINT_APPCALLBACK** *PrTimer0Callback***);**

Description

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

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId***, uint32** *u32ItemBitmap***);**

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

> **Note:** The function **u8AHI_TimerFired()** should not be called within the Timer callback function - for more information, refer to the function description on page 296.

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

PrTimer0Callback	Pointer to callback function to be registered

Returns

None

vAHI_Timer1RegisterCallback

void vAHI_Timer1RegisterCallback(
PR_HWINT_APPCALLBACK *PrTimer1Callback***);**

Description

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

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId***, uint32** *u32ItemBitmap***);**

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

> **Note:** The function **u8AHI_TimerFired()** should not be called within the Timer callback function - for more information, refer to the function description on page 296.

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

PrTimer1Callback	Pointer to callback function to be registered

Returns

None

vAHI_Timer2RegisterCallback

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

Description

This function registers a user-defined callback function that will be called when the Timer 2 (PWM1) interrupt is triggered.

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId*, **uint32** *u32ItemBitmap***);**

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

> **Note:** The function **u8AHI_TimerFired()** should not be called within the Timer callback function - for more information, refer to the function description on page 296.

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

PrTimer2Callback	Pointer to callback function to be registered

Returns

None

vAHI_Timer3RegisterCallback

void vAHI_Timer3RegisterCallback(
PR_HWINT_APPCALLBACK *PrTimer3Callback***);**

Description

This function registers a user-defined callback function that will be called when the Timer 3 (PWM2) interrupt is triggered.

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId***, uint32** *u32ItemBitmap***);**

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

> **Note:** The function **u8AHI_TimerFired()** should not be called within the Timer callback function - for more information, refer to the function description on page 296.

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

PrTimer3Callback	Pointer to callback function to be registered

Returns

None

vAHI_Timer4RegisterCallback

```
void vAHI_Timer4RegisterCallback(
              PR_HWINT_APPCALLBACK PrTimer4Callback);
```

Description

This function registers a user-defined callback function that will be called when the Timer 4 (PWM3) interrupt is triggered.

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId*, **uint32** *u32ItemBitmap***);**

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

Note: The function **u8AHI_TimerFired()** should not be called within the Timer callback function - for more information, refer to the function description on page 296.

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

PrTimer4Callback Pointer to callback function to be registered

Returns

None

vAHI_Timer5RegisterCallback

void vAHI_Timer5RegisterCallback(
PR_HWINT_APPCALLBACK *PrTimer5Callback***);**

Description

This function registers a user-defined callback function that will be called when the Timer 5 (PWM4) interrupt is triggered.

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId***, uint32** *u32ItemBitmap***);**

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

Note: The function **u8AHI_TimerFired()** should not be called within the Timer callback function - for more information, refer to the function description on page 296.

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

PrTimer5Callback	Pointer to callback function to be registered

Returns

None

vAHI_Timer6RegisterCallback

void vAHI_Timer6RegisterCallback(
PR_HWINT_APPCALLBACK *PrTimer6Callback***);**

Description

This function registers a user-defined callback function that will be called when the Timer 6 (PWM5) interrupt is triggered.

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId*, **uint32** *u32ItemBitmap***);**

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

Note: The function **u8AHI_TimerFired()** should not be called within the Timer callback function - for more information, refer to the function description on page 296.

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

PrTimer6Callback Pointer to callback function to be registered

Returns

None

vAHI_Timer7RegisterCallback

```
void vAHI_Timer7RegisterCallback(
            PR_HWINT_APPCALLBACK PrTimer7Callback);
```

Description

This function registers a user-defined callback function that will be called when the Timer 7 (PWM6) interrupt is triggered.

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId*, **uint32** *u32ItemBitmap***);**

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

> **Note:** The function **u8AHI_TimerFired()** should not be called within the Timer callback function - for more information, refer to the function description on page 296.

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

PrTimer7Callback	Pointer to callback function to be registered

Returns

None

vAHI_Timer8RegisterCallback

void vAHI_Timer8RegisterCallback(
PR_HWINT_APPCALLBACK *PrTimer8Callback***);**

Description

This function registers a user-defined callback function that will be called when the Timer 8 (Analogue Peripheral Timer) interrupt is triggered.

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId*, **uint32** *u32ItemBitmap***);**

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

> **Note:** The function **u8AHI_TimerFired()** should not be called within the Timer callback function - for more information, refer to the function description on page 296.

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

PrTimer8Callback Pointer to callback function to be registered

Returns

None

boAHI_RegisterTimerCallbackById

```
bool_t boAHI_RegisterTimerCallbackById(
        uint8 u8Timer,
        PR_HWINT_APPCALLBACK prTimerCallback);
```

Description

This function registers a user-defined callback function that will be called when the specified Timer interrupt is triggered. The function can be used instead of the dedicated callback registration functions for the individual timers.

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId*, **uint32** *u32ItemBitmap***);**

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

> **Note:** The function **u8AHI_TimerFired()** should not be called within the Timer callback function - for more information, refer to the function description on page 296.

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

u8Timer	Identity of timer: E_AHI_TIMER_0 (Timer 0) E_AHI_TIMER_1 (Timer 1) E_AHI_TIMER_2 (PWM1) E_AHI_TIMER_3 (PWM2) E_AHI_TIMER_4 (PWM3) E_AHI_TIMER_5 (PWM4) E_AHI_TIMER_6 (PWM5) E_AHI_TIMER_7 (PWM6) E_AHI_TIMER_8 (APT)
prTimerCallback	Pointer to callback function to be registered

Returns

TRUE if registration is accepted, FALSE if *u8Timer* is out of range

24. Wake Timer Functions

This chapter details the functions for controlling the wake timers. The JN517x microcontroller includes two wake timers, denoted Wake Timer 0 and Wake Timer 1, where each is a 41-bit counter.

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

The wake timers run at a nominal 32kHz, being driven from the 32kHz clock. This clock can be sourced internally or externally, as described in Section 3.1.5 (this clock selection is preserved during sleep). If sourced from the internal RC oscillator, the wake timers may run up to 18% fast or slow, depending on temperature, supply voltage and manufacturing tolerance. To achieve more accurate timings in this case, the self-calibration facility should be used to measure the 32kHz clock against the peripheral clock, which should be running at 16MHz with the system clock sourced from the external crystal oscillator.

> **Note:** For guidance on using the Wake Timer functions in JN517x application code, refer to Chapter 8.

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

vAHI_WakeTimerEnable

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

Description

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

The interrupt configuration specified using this function will take effect the next time the wake timer is started. The wake timer can be started using the function **vAHI_WakeTimerStartLarge()**.

Note that:

- If the wake timer interrupt is enabled and the timer is started, the device will be woken if the wake timer expires during sleep
- If the wake timer interrupt is disabled and the timer is started, the device will not be woken if the wake timer expires during sleep

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

Parameters

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

Returns

None

vAHI_WakeTimerStart

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

Description

This function starts the specified wake timer running. The timer runs at approximately 32kHz. Therefore, a *u32Count* value of 64000 would set the timer for approximately 2 seconds, depending on the accuracy of the 32kHz clock. The period can be made more accurate by obtaining a calibration value for the 32kHz clock and adjusting the *u32count* value accordingly (see **u32AHI_WakeTimerCalibrateEnhanced()**).

The function assumes that the timer was not already running.

If the *u8Timer* identifier is invalid, the routine exits immediately.

Parameters

u8Timer	Identity of timer: E_AHI_WAKE_TIMER_0 (Wake Timer 0) E_AHI_WAKE_TIMER_1 (Wake Timer 1)
u64Count	Number of 32kHz ticks to count for

Returns

None

u32AHI_WakeTimerRead

uint32 u32AHI_WakeTimerRead(uint8 *u8Timer***);**

Description

This function obtains the least significant 32 bits of the current count of the specified wake timer. The function should be used when the timer has been started using **vAHI_WakeTimerStart()**.

Parameters

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

Returns

0 if *u8Timer* contains an invalid wake timer

Otherwise, the 32-bit count value contained in the specified wake timer's least significant 32 bits

vAHI_WakeTimerStop

void vAHI_WakeTimerStop(uint8 *u8Timer*);

Description

This function stops the specified wake timer. Note that no interrupt will be generated.

Parameters

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

Returns

None

vAHI_WakeTimerStartLarge

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

Description

This function starts the specified wake timer with the specified count value. The wake timer will count down from this value, which is set according to the desired timer duration. On reaching zero, the timer 'fires', rolls over to 0x1FFFFFFFFFF and continues to count down.

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

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

If the 32kHz clock, which drives the wake timer, is sourced from the internal 32kHz RC oscillator then the wake timer may run up to 40% fast or 10% slow on the JN517x device. For accurate timings in this case, you are advised to first calibrate the clock using the function **u32AHI_WakeTimerCalibrate()** and adjust the specified count value accordingly.

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

Parameters

u8Timer	Identity of timer: E_AHI_WAKE_TIMER_0 (Wake Timer 0) E_AHI_WAKE_TIMER_1 (Wake Timer 1)
u64Count	Count value in 32kHz periods, i.e. 32 is 1 millisecond (this value must not exceed 0x1FFFFFFFFFF, and the values 0 and 1 must not be used)

Returns

None

u64AHI_WakeTimerReadLarge

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

Description

This function obtains the current value of the specified wake timer counter (which counts down), without stopping the counter.

Note that on reaching zero, the timer 'fires', rolls over to 0x1FFFFFFFFFF and continues to count down. The count value obtained using this function then allows the application to calculate the time that has elapsed since the wake timer fired.

The function should be used to read a counter that has been started using **vAHI_WakeTimerStartLarge()**, which will have enabled the higher 9 bits of the counter.

Parameters

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

Returns

Current value of wake timer 41-bit counter

u8AHI_WakeTimerStatus

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

Description

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

Note that a wake timer remains active after its count-down has reached zero (when the timer rolls over to 0x1FFFFFFFFFF and continues to count down).

Parameters

None

Returns

Bitmap:

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

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

u8AHI_WakeTimerFiredStatus

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

Description

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

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

Parameters

None

Returns

Bitmap:

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

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

u32AHI_WakeTimerCalibrate

uint32 u32AHI_WakeTimerCalibrate(void);

Description

This function requests a calibration of the 32kHz clock (on which the wake timers run) against the more accurate peripheral clock which must be running at 16MHz (i.e. the system clock is sourced from the external crystal oscillator). This calibration may be required if the 32kHz clock is sourced from the internal 32kHz RC oscillator - see Section 8.2.

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

The returned result, n, is interpreted as follows:

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

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

Parameters

None

Returns

Calibration measurement, n (see above)

u32AHI_WakeTimerCalibrateEnhanced

```
uint32 u32AHI_WakeTimerCalibrateEnhanced(
                                  uint16 u16CalValue);
```

Description

This function requests a calibration of the 32kHz clock (on which the wake timers run) against the more accurate peripheral clock which must be running at 16MHz (i.e. the system clock is sourced from the external crystal oscillator). This calibration may be required if the 32kHz clock is sourced from the internal 32kHz RC oscillator - see Section 8.2.

The function takes an input *u16CalValue* that is the number of 32kHz ticks over which the calibration will be performed. The function outputs the corresponding number of 16MHz ticks that have been measured over this period. If the 32kHz clock is running accurately then the output value will be equal to *u16CalValue* x 500.

The returned result, n, is interpreted as follows:

- n = *u16CalValue* x 500 ⇒ clock running at 32kHz
- n > *u16CalValue* x 500 ⇒ clock running slower than 32kHz
- n < *u16CalValue* x 500 ⇒ clock running faster than 32kHz

The returned value can be used to adjust the time interval value used to program a wake timer. If the required timer duration is T seconds, the count value N that must be specified in **vAHI_WakeTimerStartLarge()** is given by:

$$N = (u16CalValue \times 500/n) \times 32000 \times T$$

The calibration process requires the CPU to enter doze mode with interrupts disabled (necessary to stop anything other than the wake timer from waking the processor). The larger *u16CalValue* is, the longer the CPU will be unable to perform any other task. However, the best calibration accuracy will result from the largest number of 32kHz ticks that can be used.

Parameters

u16CalValue	Number of 32kHz ticks over which the calibration will be run

Returns

Calibration measurement, n (see above).

This is the measured number of 16MHz ticks corresponding to the number of 32kHz ticks provided in the input parameter.

25. Tick Timer Functions

This chapter details the functions for controlling the Tick Timer on the JN517x microcontrollers - this is a hardware timer, derived from the processor free-running clock. It can be used to generate timing interrupts to software.

The Tick Timer can be used to implement:

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

Note 1: For guidance on using the Tick Timer functions in JN517x application code, refer to Chapter 9.

Note 2: For high-precision Tick Timer operation, the peripheral clock should run at 16MHz with the system clock sourced from the external crystal oscillator. For system clock information, refer to Section 3.1.

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

vAHI_TickTimerConfigure

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

Description

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

The Tick Timer counts downwards from a reference value until the count reaches zero. This function determines what the timer will do once a count of zero has been reached. The options are

- Restart the count from the reference value
- Stop counting (single-shot mode)

The reference count is set using the function **vAHI_TickTimerInterval()**. An interrupt can be enabled which is generated on the timer transitioning from 1 to 0 - see the description of **vAHI_TickTimerIntEnable()**.

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

1. Call **vAHI_TickTimerStop()** to stop the Tick Timer.
2. Call **vAHI_TickTimerConfigure()** to set the operational mode of the Tick Timer.
3. Call **vAHI_TickTimerInterval()** to set the reference count.
4. Call **vAHI_TickTimerInit()** to start the Tick Timer in the desired mode.

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

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

Parameters

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

Returns

None

vAHI_TickTimerInterval

void vAHI_TickTimerInterval(uint32 *u32Interval***);**

Description

This function sets the 24-bit reference count for the Tick Timer.

This is the value from which the Tick Timer counts down. The action taken when the count reaches zero is determined using the function **vAHI_TickTimerConfigure()**. An interrupt can be also enabled which is generated on reaching a count of zero - see the function v**AHI_TickTimerIntEnable()**.

Parameters

u32Interval	Tick Timer reference count (in the range 0 to 0x00FFFFFF)

Returns

None

vAHI_TickTimerInit

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

Description

This function initialises the Tick Timer as follows:

- registers a user-defined callback function that will be called when the Tick Timer interrupt is triggered
- enables the interrupt
- sets the source clock to Internal (i.e. 32 MHz clock)
- starts the timer running

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

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

Interrupt handling is described in Appendix A.

Parameters

prTickTimerCallback	Pointer to callback function to be registered

Returns

None

vAHI_TickTimerClear

```
void vAHI_TickTimerClear(void);
```

Description

This function sets the current value of the Tick Timer to 0.

Parameters

None

Returns

None

u32AHI_TickTimerRead

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

Description

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

Parameters

None

Returns

Value of the Tick Timer counter

vAHI_TickTimerStop

```
void vAHI_TickTimerStop(void);
```

Description

This function stops (disables) the Tick Timer.

Parameters

None

Returns

None

bAHI_TickTimerWrapStatus

```
bool_t  bAHI_TickTimerWrapStatus(void);
```

Description

This function returns the status of the Tick Timer count, specifically whether it has counted from 1 to 0.

Parameters

None

Returns

TRUE if the Tick Timer counter has counted from 1 to 0, FALSE otherwise

vAHI_SetTickTimerSource

void vAHI_SetTickTimerSource(bool_t *boInternalClock*);

Description

This function selects the source of the clock to be used by the Tick Timer.

Parameters

boInternalClock	Source clock to be used: TRUE - The internal clock FALSE - An external clock

Returns

None

vAHI_TickTimerIntEnable

void vAHI_TickTimerIntEnable(bool_t *bIntEnable***);**

Description

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

A user-defined callback function, which is invoked when the interrupt is generated, can be registered using the function **vAHI_TickTimerInit()**.

Note that Tick Timer interrupts can be used to wake the CPU from Doze mode.

Parameters

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

Returns

None

bAHI_TickTimerIntStatus

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

Description

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

Parameters

None

Returns

TRUE if an interrupt is pending, FALSE otherwise

vAHI_TickTimerIntPendClr

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

Description

This function clears any pending Tick Timer interrupt.

Parameters

None

Returns

None

26. Watchdog Timer Functions

This chapter describes the functions for configuring and controlling the Watchdog Timer on the JN517x microcontroller.

Note: For information on the Watchdog Timer and guidance on using the Watchdog Timer functions in JN517x application code, refer to Chapter 10.

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

vAHI_WatchdogStart

```
void vAHI_WatchdogStart(uint8 u8Prescale);
```

Description

This function starts the Watchdog Timer and sets the timeout period. Note that the Watchdog Timer is enabled by default and is run with the maximum possible timeout period of 16392ms. If this function is called while the Watchdog Timer is running, it allows the timer to continue uninterrupted but modifies the timeout period.

The timeout period of the Watchdog Timer is determined by an index, specified through the parameter *u8Prescale*, and is calculated according to the formulae:

Timeout Period = 8ms if *u8Prescale* = 0

Timeout Period = $[2^{(Prescale - 1)} + 1] \times 8$ms if $1 \leq u8Prescale \leq 12$

If the Watchdog Timer is sourced from an internal RC oscillator, the actual timeout period obtained may be up to 18% less than the calculated value due to variations in the oscillator.

Be sure to set the Watchdog timeout period to be greater than the worst-case Flash memory read-write cycle. If the Watchdog times out during a Flash memory access, the JN517x microcontroller will enter programming mode. For information on read-write cycle times, refer to the relevant Flash memory data sheet.

Note that the Watchdog Timer will continue to run during Doze mode but not during Sleep or Deep Sleep mode, or when the hardware debugger has taken control of the CPU (it will, however, automatically restart using the same prescale value when the debugger un-stalls the CPU).

Parameters

u8Prescale	Index in the range 0 to 12, which determines the Watchdog timeout period (see above formulae) - gives timeout periods in the range 8 to 16392ms

Returns

None

vAHI_WatchdogStop

```
void vAHI_WatchdogStop(void);
```

Description

This function stops the Watchdog Timer and freezes the timer count.

Parameters

None

Returns

None

vAHI_WatchdogRestart

```
void vAHI_WatchdogRestart(void);
```

Description

This function re-starts the Watchdog Timer from the beginning of the timeout period.

Parameters

None

Returns

None

u16AHI_WatchdogReadValue

```
uint16 u16AHI_WatchdogReadValue(void);
```

Description

This function obtains an indication of the progress of the Watchdog Timer towards its timeout period.

The returned value is an integer in the range 0 to 255, where:

- 0 indicates that the timer has just started a new count
- 255 indicates that the timer has almost reached the timeout period

Thus, each increment of the returned value represents 1/256 of the Watchdog period - for example, a reported value of 128 indicates that the timer is about half-way through its count.

If this function is called on a transition (increment) of the Watchdog counter, the result will be unreliable. You are therefore advised to call this function repeatedly until two consecutive results are the same.

Tip: This function is useful during code development and debug to ensure that the application does not reset the Watchdog Timer too close to the Watchdog timeout period. The function should not be needed in the final application.

Parameters

None

Returns

Integer value in the range 0 to 255, indicating the progress of the Watchdog Timer

bAHI_WatchdogResetEvent

```
bool_t bAHI_WatchdogResetEvent(void);
```

Description

This function determines whether the last device reset was caused by a Watchdog Timer expiry event.

Parameters

None

Returns

TRUE if a reset occurred due to a Watchdog event, FALSE otherwise

vAHI_WatchdogException

void vAHI_WatchdogException(bool_t *bEnable***);**

Description

This function can be used to enable (or disable) an exception that will be invoked when the Watchdog Timer expires. The Watchdog exception is serviced by the stack overflow exception handler, which can call **bAHI_WatchdogResetEvent()** to determine if the Watchdog exception occurred.

If Watchdog exception handling is not enabled using this function, then the JN517x will be reset when the Watchdog Timer expires. The exception handling option is provided to allow debug on a Watchdog timeout during application development.

The stack overflow exception handler function should first be developed before enabling the Watchdog exception option.

Note: The stack overflow exception handler function should have the following prototype definition:

PUBLIC void vException_StackOverflow(void);

We would not expect an exception handler written in C to return - once it has performed any actions, it should either sit in a loop or reset the device.

Parameters

bEnable	Enable/disable exception handling: TRUE to enable FALSE to disable (default)

Returns

None

27. Pulse Counter Functions

This chapter details the functions for controlling and monitoring the pulse counters on the JN517x device. A pulse counter detects and counts pulses on an external signal that is input on an associated DIO pin.

Two 16-bit pulse counters are provided on the JN517x device, Pulse Counter 0 and Pulse Counter 1. The two counters can be combined together to provide a single 32-bit counter, if desired.

Note: For information on the pulse counters and guidance on using the Pulse Counter functions in JN517x application code, refer to Chapter 11.

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

bAHI_PulseCounterConfigure

```
bool_t bAHI_PulseCounterConfigure(uint8 u8Counter,
                                  bool_t bEdgeType,
                                  uint8 u8Debounce,
                                  uint8 u8Combine,
                                  bool_t bIntEnable);
```

Description

This function configures the specified pulse counter. The input signal will be taken from the DIO pin selected using **vAHI_SetDIOpinMultiplexValue()** prior to calling this function. The input signal for the combined pulse counter can be taken from any of the alternative input DIOs.

The following features are configured:

- **Edge detected** (*bEdgeType*): The counter can be configured to detect a pulse on its rising edge (low-to-high transition) or falling edge (high-to-low transition).
- **Debounce** (*u8Debounce*): This feature can be enabled so that a number of identical consecutive input samples are required before a change in the input signal is recognised. When disabled, the device can sleep with the 32kHz oscillator off.
- **Combined counter** (*u8Combine*): The two 16-bit pulse counters can be combined into a single 32-bit pulse counter. The combined counter is configured according to the Pulse Counter 0 settings (the Pulse Counter 1 settings are ignored) but the input signal can be taken from the input pin for either counter.
- **Interrupts** (*bIntEnable*): Interrupts can be configured to occur when the count passes a reference value, specified using **bAHI_SetPulseCounterRef()**. These interrupts are handled as System Controller interrupts by the callback function registered with **vAHI_SysCtrlRegisterCallback()** - also refer to Appendix A.

Parameters

u8Counter	Identity of pulse counter: E_AHI_PC_0 (Pulse Counter 0 or combined counter) E_AHI_PC_1 (Pulse Counter 1)
bEdgeType	Edge type on which pulse detected (and count incremented): 0: Rising edge (low-to-high transition) 1: Falling edge (high-to-low transition)
u8Debounce	Debounce setting - number of identical consecutive input samples before change in input value is recognised: 0: No debounce (maximum input frequency of 100kHz) 1: 2 samples (maximum input frequency of 3.7kHz) 2: 4 samples (maximum input frequency of 2.2kHz) 3: 8 samples (maximum input frequency of 1.2kHz)
u8Combine	Enable/disable combined 32-bit counter: E_AHI_PC_COMBINE_OFF (0 - Pulse counters not combined) E_AHI_PC_COMBINE_ON0 (1 - Counters combined using PC0 input) E_AHI_PC_COMBINE_ON1 (2 - Counters combined using PC1 input)
bIntEnable	Enable/disable pulse counter interrupts: TRUE - Enable interrupts FALSE - Disable interrrupts

Returns

TRUE if valid pulse counter specified, FALSE otherwise

bAHI_SetPulseCounterRef

bool_t bAHI_SetPulseCounterRef(uint8 *u8Counter***,**
uint32 *u32RefValue***);**

Description

This function can be used to set the reference value for the specified pulse counter.

If pulse counter interrupts are enabled through **bAHI_PulseCounterConfigure()**, an interrupt will be generated when the counter passes the reference value - that is, when the count reaches *(reference value + 1)*. This value is retained during sleep and, when generated, the pulse counter interrupt can wake the device from sleep.

The reference value must be 16-bit when specified for the individual pulse counters, but can be a 32-bit value when specified for the combined counter (enabled through **bAHI_PulseCounterConfigure()**). The reference value can be modified at any time.

The pulse counter can increment beyond its reference value and when it reaches its maximum value (65535, or 4294967295 for the combined counter), it will wrap around to zero.

Parameters

u8Counter	Identity of pulse counter: E_AHI_PC_0 (Pulse Counter 0 or combined counter) E_AHI_PC_1 (Pulse Counter 1)
u32RefValue	Reference value to be set - as a 16-bit value, it must be specified in the lower 16 bits of this 32-bit parameter, unless for the combined counter when a full 32-bit value should be specified

Returns

TRUE if valid pulse counter and reference count

FALSE if invalid pulse counter or reference count (>16 bits for single counter)

bAHI_StartPulseCounter

bool_t bAHI_StartPulseCounter(uint8 *u8Counter*);

Description

This function starts the specified pulse counter.

Note that the count may increment by one when this function is called (even though no pulse has been detected).

Parameters

u8Counter	Identity of pulse counter: E_AHI_PC_0 (Pulse Counter 0 or combined counter) E_AHI_PC_1 (Pulse Counter 1)

Returns

TRUE if valid pulse counter has been specified and started, FALSE otherwise

bAHI_StopPulseCounter

bool_t bAHI_StopPulseCounter(uint8 *u8Counter***);**

Description

This function stops the specified pulse counter.

Note that the count will freeze when this function is called. Thus, this count can subsequently be read using **bAHI_Read16BitCounter()** or **bAHI_Read32BitCounter()** for the combined counter.

Parameters

u8Counter	Identity of pulse counter: E_AHI_PC_0 (Pulse Counter 0 or combined counter) E_AHI_PC_1 (Pulse Counter 1)

Returns

TRUE if valid pulse counter has been specified and stopped, FALSE otherwise

u32AHI_PulseCounterStatus

```
uint32 u32AHI_PulseCounterStatus(void);
```

Description

This function obtains the status of the pulse counters. It can be used to check whether the pulse counters have reached their reference values (set using the function **bAHI_SetPulseCounterRef()**).

The status of each pulse counter is returned by this function in a 32-bit bitmap value - bit 22 for Pulse Counter 0 and bit 23 for Pulse Counter 1. If the combined pulse counter is in use, its status is returned through bit 22.

If a pulse counter has reached its reference value then once the function has returned this status, the internal status bit is cleared for the corresponding pulse counter.

The function can be used to poll the pulse counters. Alternatively, interrupts can be enabled (through **bAHI_PulseCounterConfigure()**) that are generated when the pulse counters pass their reference values.

Parameters

None

Returns

32-bit value in which bit 23 indicates the status of Pulse Counter 1 and bit 22 indicates the status of Pulse Counter 0 or the combined counter. The bit values are interpreted as follows:

1 - pulse counter has reached its reference value
0 - pulse counter is still counting or is not in use

bAHI_Read16BitCounter

```
bool_t bAHI_Read16BitCounter(uint8 u8Counter,
                             uint16 *pu16Count);
```

Description

This function obtains the current count of the specified 16-bit pulse counter, without stopping the counter or clearing the count.

Note that this function can only be used to read the value of an individual 16-bit counter (Pulse Counter 0 or Pulse Counter 1) and cannot read the value of the combined 32-bit counter. If the combined counter is in use, its count value can be obtained using the function **bAHI_Read32BitCounter()**.

Parameters

u8Counter	Identity of pulse counter: E_AHI_PC_0 (Pulse Counter 0) E_AHI_PC_1 (Pulse Counter 1)
**pu16Count*	Pointer to location to receive 16-bit count

Returns

TRUE if valid pulse counter specified, FALSE otherwise

bAHI_Read32BitCounter

bool_t bAHI_Read32BitCounter(uint32 ****pu32Count*);**

Description

This function obtains the current count of the combined 32-bit pulse counter, without stopping the counter or clearing the count.

Note that this function can only be used to read the value of the combined 32-bit pulse counter and cannot read the value of a 16-bit pulse counter used in isolation. The returned Boolean value of this function indicates if the pulse counters have been combined. If the combined counter is not use, the count value of an individual 16-bit pulse counter can be obtained using the function **bAHI_Read16BitCounter()**.

Parameters

**pu32Count*	Pointer to location to receive 32-bit count

Returns

TRUE if combined 32-bit counter in use, FALSE otherwise

bAHI_Clear16BitPulseCounter

bool_t bAHI_Clear16BitPulseCounter(uint8 const *u8Counter***);**

Description

This function clears the count of the specified 16-bit pulse counter.

Note that this function can only be used to clear the count of an individual 16-bit counter (Pulse Counter 0 or Pulse Counter 1) and cannot clear the count of the combined 32-bit counter. To clear the latter, use the function **bAHI_Clear32BitPulseCounter()**.

Parameters

u8Counter	Identity of pulse counter: E_AHI_PC_0 (Pulse Counter 0) E_AHI_PC_1 (Pulse Counter 1)

Returns

TRUE if valid pulse counter specified, FALSE otherwise

bAHI_Clear32BitPulseCounter

bool_t bAHI_Clear32BitPulseCounter(void);

Description

This function clears the count of the combined 32-bit pulse counter.

Note that this function can only be used to clear the count of the combined 32-bit pulse counter and cannot clear the count of a 16-bit pulse counter used in isolation. To clear the latter, use the function **bAHI_Clear16BitPulseCounter()**.

Parameters

None

Returns

TRUE if combined 32-bit counter in use, FALSE otherwise

28. Infra-Red Transmitter Functions

This chapter details the functions for controlling and monitoring the infra-red transmitter on the JN517x device. Infra-red transmission is a special feature of Timer PWM4 in which the timer is used to generate a carrier waveform that is modulated by a programmable bit sequence and output on the associated Timer PWM4 output pin.

Note: For information and guidance on using the infra-red transmitter functions in JN517x application code, refer to Chapter 12.

The Infra-Red Transmitter functions are listed below, along with their page references:

bAHI_InfraredEnable

```
bool_t bAHI_InfraredEnable(
        uint8 u8Prescale,
        uint16 u16Hi,
        uint16 u16Lo,
        uint16 u16BitPeriodInCarrierPeriods,
        bool_t bInvertOutput,
        bool_t bInterruptEnable);
```

Description

This function enables PWM Timer 4 (Timer 5) for infra-red transmission and configures the carrier waveform, data-bit period, output polarity and interrupt behaviour. The function must be initially called before any of the other functions is this chapter.

> **Note:** If enabling infra-red transmission, none of the Timer functions listed in Chapter 23 should be called for Timer 5.

A single interrupt can be enabled to indicate the end of transmission. This interrupt is handled as an Infra-Red Transmitter interrupt by the callback function registered with **vAHI_InfraredRegisterCallback()** - also refer to Appendix A.

Before operating the Infra-Red Transmitter, ensure that the PWM4/IR output signal is routed to DIO3 using the function **vAHI_SetDIOpinMultiplexValue()**.

Parameters

u8Prescale	Prescale index (0 to 16) used to divide down the peripheral clock to produce the timer clock (divider is $2^{u8Prescale}$)
u16Hi	Number of clock periods after starting the timer before the carrier goes high (i.e. carrier low duration)
u16Lo	Number of clock periods after starting the timer before the carrier goes low again (i.e. carrier period)
u16BitPeriodsInCarrierPeriods	Bit-period in units of the carrier period (1 to 256)
bInvertOutput	Output polarity: TRUE - Output polarity is inverted FALSE - Output polarity is non-inverted
bInterruptEnable	Enable/disable infra-red transmitter interrupt: TRUE - Enable interrupt FALSE - Disable interrrupt

Returns

TRUE if parameters are valid

FALSE otherwise

vAHI_InfraredDisable

```
void vAHI_InfraredDisable(void);
```

Description

This function can be used to disable PWM Timer 4 from infra-red transmission. If required, this function must be called after **bAHI_InfraredEnable()**.

Parameters

None

Returns

None

bAHI_InfraredStart

```
bool_t bAHI_InfraredStart(
                    uint32 *pu32BufferAddress,
                    uint16 u16TransmissionLengthInBits);
```

Description

This function is used to start the infra-red transmission of a programmed bit-sequence stored in a 32-bit wide data array (i.e. transmit buffer). This function should be called after **bAHI_InfraredEnable()**.

Parameters

**pu32BufferAddress*	Pointer to start of transmit buffer in RAM
u16TransmissionLengthInBits	Length of transmission sequence in bits (1 to 4096)

Returns

TRUE if transmission will start (due to valid input parameter values)

FALSE if transmission will not start (due to invalid input parameter values)

bAHI_InfraredStatus

```
bool_t bAHI_InfraredStatus(void);
```

Description

This function can be used to check the status of the infra-red transmission. If required, this function must be called after **bAHI_InfraredEnable()**.

Parameters

None

Returns

TRUE if a transmission is in progress

FALSE if a transmission is not (or no-longer) in progress

vAHI_InfraredRegisterCallback

**void vAHI_InfraredRegisterCallback(
PR_HWINT_APPCALLBACK** *prInfraredCallback***);**

Description

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

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId***, uint32** *u32ItemBitmap***);**

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

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

prInfraredCallback Pointer to callback function to be registered

Returns

None

29. I^2C Interface Functions

This chapter details the functions for controlling the Inter-Integrated Circuit (I^2C) interface on the JN517x microcontroller. The I^2C interface is logic-compatible with similar interfaces such as the 2-wire Serial Interface of the JN516x and SMbus.

Two sets of functions are described in this chapter, one set for an I^2C master and another set for an I^2C slave:

- Functions for controlling the I^2C master are described in Section 29.2.
- Functions for controlling the I^2C slave are described in Section 29.3.

General functions that apply to both I^2C master and I^2C slave modes are described in Section 29.1.

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

Note: For guidance on using the I^2C Interface functions in JN517x application code, refer to Chapter 13.

29.1 I^{2}C Common Functions

This section contains the functions common to both the Master and Slave modes of the I^{2}C interface

The I^{2}C Common functions are listed below, along with their page references:

vAHI_I2CcontrollerConfigure

```
void vAHI_I2CcontrollerConfigure(
        bool_t boEnableDataInputSpikeSuppressionFilter,
        bool_t boEnableClockInputSpikeSuppressionFilter,
        bool_t boDisableSlewControlForDataPin,
        bool_t boDisableSlewControlForClockPin,
        bool_t boUseDIOpins4and5OpenDrain);
```

Description

This function configures the I²C controller band and selects the DIO pins through which the SDA and SCL signals will be routed. The pins for SDA and SCL may be respectively selected as:

- DIO5 and DIO4 - these are enabled as open-drain pins
- DIO2 and DIO3 - these are normal DIO pins

The function can also be used to enable/disable the following features on the individual DIO pins selected for I²C use:

- spike suppression, which adds filtering that removes spikes of up to 60ns duration
- a slew rate limit for the signals, in line with the I²C specification

Parameters

boEnableDataInputSpikeSuppressionFilter

TRUE - Enable the spike suppression filter on the SDA input

FALSE - Disable the spike suppression filter on the SDA input

boEnableClockInputSpikeSuppressionFilter

TRUE - Enable the spike suppression filter on the SCL input

FALSE - Disable the spike suppression filter on the SCL input

boDisableSlewControlForDataPin

TRUE - Disable slew control on SDA pin

FALSE - Enable slew control on SDA pin

boDisableSlewControlForClockPin

TRUE - Disable slew control on SCL pin

FALSE - Enable slew control on SCL pin

boUseDIOpins4and5OpenDrain

TRUE - use DIO5 and DIO4 pins, which are open-drain

FALSE - use DIO2 and DIO3 pins, which are normal DIOs

Returns

None

vAHI_SetI2CpinsToMFIOmode

void vAHI_SetI2CpinsToMFIOmode(void);

Description

This function allow the pins DIO4 and DIO5 to be configured as Multi-Function IO (MFIO) pins rather than used for specialised I^2C IO cells (as they are configured by default from reset). The function disables the open drain, spike suppression and slew rate functionality that is needed when the pins are used as true I^2C signals.

This function is called as part of **vAHI_Init()** so that all DIO pins are configured for MFIO functionality at initialisation time. When the I^2C operation is required, the functionality is explicitly enabled using **vAHI_I2CcontrollerConfigure()**.

If I^2C functionality has been used on DIO4 and DIO5, and then switched off and the DIO pins reassigned to another use, this function should be called to ensure that the pins are configured for MFIO signals. Failure to do this may result in incorrect logic levels on the IO pins due to the open-drain IO cells remaining enabled.

Parameters

None

Returns

None

u32AHI_I2CGetStatus

```
uint32 u32AHI_I2CGetStatus(void);
```

Description

This function returns the status of the I^2C interface.

Parameters

None

Returns

Bitmap containing status information, as follows:

Bit	Name	Description
31-16	-	Reserved
11	TX_FIFO_block	I^2C access to the TX_FIFO is blocked. When set, a read attempt (from the I^2C side) will result in the I^2C clock being stretched. Set upon detection of a STOP or RESTART (Slave transmitter mode). Cleared when TX_FIFO is empty.
10	RX_FIFO_block	I^2C access to the RX_FIFO is blocked. When set, a write attempt (from the I^2C side) will result in the I^2C clock being stretched. Set upon detection of: • an Arbitration Failure (Master mode) • a Stop or Restart (Slave receiver mode) Cleared when RX_FIFO is empty.
9	TXS_FIFO_full	TXS_FIFO is full.
8	TXS_FIFO_empty	TXS_FIFO is empty.
7	TX_FIFO_full	TX_FIFO is full.
6	TX_FIFO_empty	TX_FIFO is empty.
5	RX_FIFO_full	RX_FIFO is full.
4	RX_FIFO_empty	RX_FIFO is empty.
3	SDA	Current status of SDA line.
2	SCL	Current status of SCL line.
1	I2C_Bus_Active	Set when START condition is detected, cleared when STOP condition is detected.
0	Mst_Slv_Mode	Module is in Master mode (1) or Slave mode (0).

vAHI_I2CReset

```
void vAHI_I2CReset(void);
```

Description

This function performs a soft reset of the I²C Interface.

Parameters

None

Returns

None

boAHI_I2CIsBusy

```
bool_t boAHI_I2CIsBusy(void);
```

Description

This function indicates whether the I^2C Interface on the device is busy.

Parameters

None

Returns

TRUE if the interface is busy

FALSE if the interface is not busy

vAHI_I2CRegisterCallback

void vAHI_I2CRegisterCallback(
PR_HWINT_APPCALLBACK *pfI2CCallback***);**

Description

This function is used to register a user-supplied callback routine which will be called from within the I^2C interrupt handler.

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId***, uint32** *u32ItemBitmap***);**

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

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

pfI2CCallback	Pointer to callback function to be registered

Returns

None

pfAHI_I2CGetCallback

```
PR_HWINT_APPCALLBACK pfAHI_I2CGetCallback(void);
```

Description

This function obtains a pointer to the I^2C callback function (if any) that has been registered using **vAHI_I2CRegisterCallback()**. This is useful for creating a callback function chain.

Parameters

None

Returns

Pointer to registered I^2C callback function (null pointer if one is not registered)

29.2 I^2C Master Functions

This section details the functions for controlling a I^2C Interface master on a JN517x microcontroller.

The I^2C master can implement bi-directional communication with a slave device on the I^2C bus (I^2C slave functions are also provided and are described in Section 29.3). Note that the I^2C bus on the JN517x device can have more than one master, but multiple masters cannot use the bus at the same time - to avoid this, an arbitration scheme is provided.

When enabled, this interface may use DIO3 or DIO4 as a clock (SCL) and DIO2 or DIO5 as a bi-directional data line (SDA). The DIO pins to be used are selected using **vAHI_SetDIOpinMultiplexValue()**. Note that pins DIO4 and DIO5 are true open-drain I/Os which are preferred for I^2C implementation. The clock is scaled from the peripheral clock, which must run at 16MHz with the system clock sourced from the external crystal oscillator (for system clock information, refer to Section 3.1).

The I^2C Master functions are listed below, along with their page references:

boAHI_I2CMasterConfigure

```
bool_t boAHI_I2CMasterConfigure(
                    uint32 u32SCLfrequencyInHertz);
```

Description

This function configures and enables the I^{2}C Master peripheral. Interrupts are cleared, and the Rx and Tx FIFOs are initialised. The parameters allow the clock rate of the I^{2}C bus to be set.

Parameters

u32SCLfrequencyInHertz	SCL clock frequency in Hz (maximum value is 1MHz)

Returns

None

u16AHI_I2CMasterReadWriteData

```
uint16 u16AHI_I2CMasterReadWriteData(
                    uint16 u16SlaveAddress,
                    bool_t boExtendAddr,
                    uint8 *pu8MasterDataBuffer,
                    uint16 u16MasterDataBufferLength,
                    bool_t boMasterWriteData,
                    bool_t boMasterHoldBus);
```

Description

This function sends or receives a number of bytes to or from a slave device on the I^2C bus. If reading from a slave, the Master data buffer contains dummy transmit data for the transfer.

Parameters

u16SlaveAddress	The address of the slave in the transaction. Can be 7- or 10-bit address depending on *boExtendAddr* (below)
boExtendAddr	Boolean indicating length of slave address to be used: TRUE - use 10-bit slave address FALSE - use 7-bit slave address
p*u8MasterDataBuffer*	Pointer to the array of uint8 used to store the data to be transmitted
u16MasterDataBufferLength	Amount of data to send/receive, in bytes (i.e. the size of the transaction payload)
boMasterWriteData	Boolean indicating the type of transaction: TRUE - Write to Slave by Master FALSE - Read from Slave by Master
boMasterHoldBus	Boolean indicating whether the bus is to be held between successive transactions (i.e. no STOP bit): TRUE - hold the bus FALSE - send STOP bit to complete the transfer

Returns

Number of bytes written/read

u16AHI_I2CMasterMultiReadWriteData

uint16 u16AHI_I2CMasterMultiReadWriteData(
tsAHII2CBCTentry ***pasBlockChainTransfer*,**
uint16 *u16NumberOfTransfers***);**

Description

This function performs a number of I^2C transfers, the details of which are held in an array of transfer descriptors pointed to by *pasBlockChainTransfer*. The number of transfers in the chain is given by *u16NumberOfTransfers*.

Each transfer descriptor consists of the information in the following structure:

```
typedef struct {
   uint16   u16SlaveAddress;
   bool_t   boExtendAddr;
   uint8   *pu8DataBuffer;
   uint16   u16DataLength;
   bool_t   boMasterWriteData;
} tsAHII2CBCTentry;
```

The fields of the above structure correspond to the parameters of the function **u16vAHI_I2CMasterReadWriteData()** - refer to the function description for details.

Parameters

pasBlockChainTransfer	Pointer to the array of transfer descriptors for the chain of transfers
u16NumberOfTransfers	Number of transfers to be performed in the chain

Returns

Zero or a positive number indicates the number of bytes written/read during the transaction chain.

On an error, a negative number is returned:

- -1 indicates a parameter error - either the pointer to the Block Chain Transfer array is null or the number of transfers is 0
- -2 indicates the transactions cannot start since there is already one in progress
- -3 indicates that an invalid data buffer has been specified using `pu8DataBuffer`

boAHI_I2CIsMasterTransferComplete

```
bool_t boAHI_I2CIsMasterTransferComplete(
                        uint32 *pu32TransferHasFailedMask);
```

Description

This function obtains the state of the current I^2C Master transfer - that is, whether the transfer has completed or is on-going. Any transfer errors will be returned in the Transfer Error Mask - enumerations are defined for the error conditions, as follows:

Enumeration	Description
REG_I2C_INT_MTNA_MASK	Master Transmitter No Acknowledge Interrupt: Master has received a NAK from the slave
REG_I2C_INT_TAF_MASK	Transmitter Arbitration Failure Interrupt: Master does not have ownership of the bus
REG_I2C_INT_IBE_MASK	I2C bus error
REG_I2C_INT_STSD_MASK	Slave TX Stop Detect Interrupt: Slave has sent a stop bit

Parameters

pu32TransferHasFailedMask	Pointer to location to receive Transfer Error Mask (can be null)

Returns

TRUE if the transfer has finished, FALSE if the transfer is still in progress

u32AHI_I2CTransferHasFailed

uint32 u32AHI_I2CTransferHasFailed(void);

Description

This function checks whether there has been any errors during an I^2C Master transfer and returns an error mask indicating the error conditions.

Parameters

None

Returns

Error mask - enumerations are defined for the error conditions, as follows:

Enumeration	Description
REG_I2C_INT_MTNA_MASK	Master Transmitter No Acknowledge Interrupt: Master has received a NAK from the slave
REG_I2C_INT_TAF_MASK	Transmitter Arbitration Failure Interrupt: Master does not have ownership of the bus
REG_I2C_INT_IBE_MASK	I2C bus error
REG_I2C_INT_STSD_MASK	Slave TX Stop Detect Interrupt: Slave has sent a stop bit

boAHI_I2CIsMaster

```
bool_t boAHI_I2CIsMaster(void);
```

Description

This function obtains the mode of the I^2C interface on the device.

Parameters

None

Returns

TRUE if the interface is a Master, FALSE if the interface is a Slave

29.3 I²C Slave Functions

This section details the functions for controlling a 2-wire I²C slave on the JN517x microcontroller.

As in the case of an I²C master, the I²C slave uses the signal I2C_SCL as a clock and signal I2C_SDA as a bi-directional data line, but does not supply the clock. The I2C_SCL signal can be routed to DIO3 or DIO4, and the I2C_SDA signal can be routed to DIO2 or DIO5.

The I²C Slave functions are listed below, along with their page references:

boAHI_I2CSlaveConfigure

```
bool_t boAHI_I2CSlaveConfigure(
        uint16 u16SlaveAddress,
        bool_t bExtendAddr,
        uint8 *pu8SlaveReceiveBuffer,
        uint16 u16SlaveReceiveBufferLength);
```

Description

This function configures the I^2C Slave interface by setting up the slave address and initialising the buffer to be used for receiving data over the bus.

Parameters

u16SlaveAddress	Slave address (7-bit or 10-bit, as defined by *bExtendAdd*)
bExtendAddr	Size of slave address (specified through *u16SlaveAddress*): TRUE - 10-bit address FALSE - 7-bit address
pu8SlaveReceiveBuffer	Pointer to array of bytes to be used as the receive buffer for the Slave
u16SlaveReceiveBufferLength	Size of the receive buffer, in bytes

Returns

TRUE if the command executed correctly, FALSE if the pointer to the receive buffer is invalid (NULL) or the receive buffer length is 0

u16AHI_I2CSlaveWriteData

```
uint16 u16AHI_I2CSlaveWriteData(
                    uint8 *pu8SlaveSendBuffer,
                    uint16 u16SlaveSendBufferLength);
```

Description

This function sends a number of bytes of data from a Slave to the Master on an I^2C bus. The data supplied to the function must be complete - even if all the data does not fit into the interface's Tx FIFO, it will be sent under interrupt control. Data is held in the Slave Send Buffer and the size of the payload is given by the parameter *u16SlaveSendBufferLength*.

Parameters

pu8SlaveSendBuffer	Pointer to array of bytes which form the Slave Send Buffer where the payload to be sent over the I^2C interface is written
u16SlaveSendBufferLength	The size of the payload to be sent. Note that the size cannot be 0, nor is there any checking of the payload size against the actual buffer size, to avoid buffer overflow. The calling routine must check this before calling the function.

Returns

0 if the function did not execute due to errors in parameters (e.g. *pu8SlaveSendBuffer* set to NULL or *u16SlaveSendBufferLength* is 0), otherwise the number of bytes sent over the I^2C bus.

30. SPI Master Functions

This chapter details the functions for controlling the Serial Peripheral Interface (SPI) master on the JN517x microcontroller. The SPI allows high-speed synchronous data transfer between the microcontroller and peripheral devices. When JN517x device operates as the master on the SPI bus, all other devices connected to the bus are expected to be slave devices under the control of the microcontroller's CPU.

Note 1: For information on the SPI master and guidance on using the SPI Master functions in JN517x application code, refer to Chapter 14.

Note 2: SPI Slave functions are detailed in Chapter 31.

Note 3: On a JN517x device, the SPI Master is disabled by default and shares its pins with other functions.

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

vAHI_SpiConfigureNoneDIO

```
void vAHI_SpiConfigureNoneDIO(
                    uint8 u8SlaveEnable,
                    bool_t bLsbFirst,
                    bool_t bPolarity,
                    bool_t bPhase,
                    uint8 u8ClockDivider,
                    bool_t bInterruptEnable,
                    bool_t bAutoSlaveSelect);
```

Description

This function configures and enables the SPI master. The signals SPI_M_SCK, SPI_M_MOSI and SPI_M_MISO are enabled and are available on the following DIO pins, as selected by **vAHI_SetDIOpinMultiplexValue()**:

Signal	Pins
SPI_M_SCK	DO0, DIO11
SPI_M_MOSI	DIO7, DIO15
SPI_M_MISO	DO1, DIO18
SPI_M_SEL0	DIO6, DIO17
SPI_M_SEL1	DIO15
SPI_M_SEL2	DIO7

The function also allows the number of SPI slaves (of the master) to be set. Three slave-select lines are available (SPI_M_SEL0/1/2), but only two slaves can be supported due to SPI_M_SEL2 sharing the same DIO pin as SPI_M_MOSI (DIO7) and SPI_M_SEL1 sharing the same pin as SPI_M_MOSI (DIO15) - since SPI_M_MOSI must be present, only one of SPI_M_SEL1 or 2 can be used.

The following features are also configurable using this function:

- Data transfer order - whether the least significant bit is transferred first or last
- Clock polarity and phase, which together determine the SPI mode (0, 1, 2 or 3) and therefore the clock edge on which data is latched:
  - SPI Mode 0: polarity=0, phase=0
  - SPI Mode 1: polarity=0, phase=1
  - SPI Mode 2: polarity=1, phase=0
  - SPI Mode 3: polarity=1, phase=1
- Clock divisor - the value used to derive the SPI clock from the peripheral clock
- SPI interrupt - generated when an API transfer has completed (note that interrupts are only worth using if the SPI clock frequency is much less than 16MHz)

- Automatic slave selection - enable the programmed slave-select line or lines (see **vAHI_SpiSelect()**) to be automatically asserted at the start of a transfer and de-asserted when the transfer completes. If not enabled, the slave-select lines will reflect the value set by **vAHI_SpiSelect()** directly.

Parameters

u8SlaveEnable	Number of SPI slaves to control. Valid values are 0-3 (higher values are truncated to 3)
bLsbFirst	Enable/disable data transfer with the least significant bit (LSB) transferred first: TRUE - enable FALSE - disable
bPolarity	Clock polarity: FALSE - unchanged TRUE - inverted
bPhase	Phase: FALSE - latch data on leading edge of clock TRUE - latch data on trailing edge of clock
u8ClockDivider	Clock divisor in the range 0 to 63. Peripheral clock is divided by 2 x *u8ClockDivider,* but 0 is a special value used when no clock division is required
bInterruptEnable	Enable/disable interrupt when an SPI transfer has completed: TRUE - enable FALSE - disable
bAutoSlaveSelect	Enable/disable automatic slave selection: TRUE - enable FALSE - disable

Note that the parameters *bPolarity* and *bPhase* are named differently in the library header file.

Returns

None

vAHI_SpiConfigure

```
void vAHI_SpiConfigure(uint8 u8SlaveEnable,
                       bool_t bLsbFirst,
                       bool_t bPolarity,
                       bool_t bPhase,
                       uint8 u8ClockDivider,
                       bool_t bInterruptEnable,
                       bool_t bAutoSlaveSelect);
```

Description

This function configures and enables the SPI master.

It performs the same operations as **vAHI_SpiConfigureNoneDIO()**, with the difference that it also routes the SPI master signals as follows:

Signal	Pin
SPI_M_SCK	DO0
SPI_M_MOSI	DIO7
SPI_M_MISO	DO1
SPI_M_SEL0	DIO6
SPI_M_SEL1	DIO15

The following features are also configurable using this function:

- Data transfer order - whether the least significant bit is transferred first or last
- Clock polarity and phase, which together determine the SPI mode (0, 1, 2 or 3) and therefore the clock edge on which data is latched:
  - SPI Mode 0: polarity=0, phase=0
  - SPI Mode 1: polarity=0, phase=1
  - SPI Mode 2: polarity=1, phase=0
  - SPI Mode 3: polarity=1, phase=1
- Clock divisor - the value used to derive the SPI clock from the peripheral clock
- SPI interrupt - generated when an API transfer has completed (note that interrupts are only worth using if the SPI clock frequency is much less than 16MHz)
- Automatic slave selection - enable the programmed slave-select line or lines (see **vAHI_SpiSelect()**) to be automatically asserted at the start of a transfer and de-asserted when the transfer completes. If not enabled, the slave-select lines will reflect the value set by **vAHI_SpiSelect()** directly.

Parameters

u8SlaveEnable	Number of SPI slaves to control. Valid values are 0-3 (higher values are truncated to 3)
bLsbFirst	Enable/disable data transfer with the least significant bit (LSB) transferred first: TRUE - enable FALSE - disable
bPolarity	Clock polarity: FALSE - unchanged TRUE - inverted
bPhase	Phase: FALSE - latch data on leading edge of clock TRUE - latch data on trailing edge of clock
u8ClockDivider	Clock divisor in the range 0 to 63. Peripheral clock is divided by 2 x *u8ClockDivider,* but 0 is a special value used when no clock division is required
bInterruptEnable	Enable/disable interrupt when an SPI transfer has completed: TRUE - enable FALSE - disable
bAutoSlaveSelect	Enable/disable automatic slave selection: TRUE - enable FALSE - disable

Note that the parameters *bPolarity* and *bPhase* are named differently in the library header file.

Returns

None

vAHI_SpiReadConfiguration

void vAHI_SpiReadConfiguration(
tSpiConfiguration **ptConfiguration***);**

Description

This function obtains the current configuration of the SPI bus.

This function is intended to be used in a system where the SPI bus is used in multiple configurations to allow the state to be restored later using the function **vAHI_SpiRestoreConfiguration()**. Therefore, no knowledge is needed of the configuration details.

Parameters

**ptConfiguration*	Pointer to location to receive obtained SPI configuration

Returns

None

vAHI_SpiRestoreConfiguration

```
void vAHI_SpiRestoreConfiguration(
                    tSpiConfiguration *ptConfiguration);
```

Description

This function restores the SPI bus configuration using the configuration previously obtained using **vAHI_SpiReadConfiguration()**.

Parameters

**ptConfiguration*	Pointer to SPI configuration to be restored

Returns

None

vAHI_SpiSelect

void vAHI_SpiSelect(uint8 *u8SlaveMask***);**

Description

This function sets the active slave-select line(s) to use.

The slave-select lines are asserted immediately if “automatic slave selection” is disabled, or otherwise only during data transfers. The number of valid bits in *u8SlaveMask* depends on the setting of *u8SlaveEnable* in a previous call to **vAHI_SpiConfigure()**, as follows:

u8SlaveEnable	Valid bits in *u8SlaveMask*
0	Bit 0
1	Bits 0, 1
2	Bits 0, 1, 2

Parameters

u8SlaveMask Bitmap - one bit per slave-select line

Returns

None

vAHI_SpiStop

```
void vAHI_SpiStop(void);
```

Description

This function clears any active slave-select lines. It has the same effect as **vAHI_SpiSelect(0)**.

Parameters

None

Returns

None

vAHI_SpiDisable

```
void vAHI_SpiDisable(void);
```

Description

This function disables the SPI Master.

Parameters

None

Returns

None

vAHI_SpiStartTransfer

void vAHI_SpiStartTransfer(uint8 *u8CharLen*, **uint32** *u32Out***);**

Description

This function can be used to start a data transfer to selected slave(s). The data length for the transfer can be specified in the range 1 to 32 bits.

It is assumed that **vAHI_SpiSelect()** has been called to set the slave(s) to communicate with. If interrupts are enabled for the SPI master, an interrupt will be generated when the transfer has completed.

The function **u32AHI_SpiReadTransfer32()** should be used to read the transferred data, with the data aligned to the right (lower bits).

Parameters

u8CharLen	Value in range 0-31 indicating data length for transfer: 0 - 1-bit data 1 - 2-bit data 2 - 3-bit data : 31 - 32-bit data
u32Out	Data to transmit, aligned to the right (e.g. for an 8-bit transfer, store the data in bits 0-7)

Returns

None

u32AHI_SpiReadTransfer32

```
uint32 u32AHI_SpiReadTransfer32(void);
```

Description

This function obtains the received data after a SPI transfer has completed that was started using **vAHI_SpiStartTransfer()** or **vAHI_SpiSetContinuous()**. The read data is aligned to the right (lower bits).

Parameters

None

Returns

Received data (32 bits)

u16AHI_SpiReadTransfer16

```
uint16 u16AHI_SpiReadTransfer16(void);
```

Description

This function obtains the received data after a 16-bit SPI transfer has completed.

Parameters

None

Returns

Received data (16 bits)

u8AHI_SpiReadTransfer8

```
uint8 u8AHI_SpiReadTransfer8(void);
```

Description

This function obtains the received data after a 8-bit SPI transfer has completed.

Parameters

None

Returns

Received data (8 bits)

vAHI_SpiContinuous

void vAHI_SpiContinuous(bool_t *bEnable*,
uint8 *u8CharLen***);**

Description

This function can be used to enable/disable continuous read mode. The function allows continuous data transfers to the SPI master and facilitates back-to-back reads of the received data. In this mode, incoming data transfers are automatically controlled by hardware - data is received and the hardware then waits for this data to be read by the software before allowing the next data transfer.

The data length for an individual transfer can be specified in the range 1 to 32 bits.

If used to enable continuous mode, the function will start the transfers (so there is no need to call a SPI start transfer function. If used to disable continuous mode, the function will stop any existing transfers (following the function call, one more transfer is made before the transfers are stopped).

To determine when data is ready to be read, the application should check whether the interface is busy by calling the function **bAHI_SpiPollBusy()**. If it is not busy receiving data, the data from the previous transfer can be read by calling **u32AHI_SpiReadTransfer32()**, with the data aligned to the right (lower bits). Once the data has been read, the next transfer will automatically occur.

Parameters

bEnable	Enable/disable continuous read mode and start/stop transfers: TRUE - enable mode and start transfers FALSE - stop transfers and disable mode
u8CharLen	Value in range 0-31 indicating data length for transfer: 0 - 1-bit data 1 - 2-bit data 2 - 3-bit data : 31 - 32-bit data

Returns

None

bAHI_SpiPollBusy

```
bool_t bAHI_SpiPollBusy(void);
```

Description

This function polls the SPI master to determine whether it is currently busy performing a data transfer.

Parameters

None

Returns

TRUE if the SPI master is performing a transfer, FALSE otherwise

vAHI_SpiWaitBusy

```
void vAHI_SpiWaitBusy(void);
```

Description

This function waits for the SPI master to complete a transfer and then returns.

Parameters

None

Returns

None

vAHI_SpiSetDelayReadEdge

void vAHI_SpiSetDelayReadEdge(bool_t *bSetDreBit***);**

Description

This function can be used to introduce a delay to the SCLK edge used to sample received data. The delay is by half a SCLK period relative to the normal position (so is the same edge used by the slave device to transmit the next data bit).

The function should be used when the round-trip delay of SCLK out to MISO IN is large compared with half a SCLK period (e.g. fast SCLK, low voltage, slow slave device), to allow a faster transfer rate to be used than would otherwise be possible.

Parameters

bSetDreBit	Enable/disable read edge delay: TRUE - enable FALSE - disable

Returns

None

vAHI_SpiSetContinuousMode

void vAHI_SpiSetContinuousMode(bool_t *bEnable*);

Description

This function allows the SPI Master to be configured to run in continuous read mode.

It should only be called when the SPI Master is not busy, in order to avoid generating a spurious transaction start.

Parameters

bEnable	Enables/disables continuous read mode: TRUE - enable continuous read mode FALSE - disable continuous read mode

Returns

None

vAHI_SpiRegisterCallback

void vAHI_SpiRegisterCallback(
PR_HWINT_APPCALLBACK *prSpiCallback***);**

Description

This function registers an application callback that will be called when the SPI interrupt is triggered.

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId***, uint32** *u32ItemBitmap***);**

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

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

prSpiCallback	Pointer to callback function to be registered

Returns

None

31. SPI Slave Functions

This chapter details the functions for controlling the Serial Peripheral Interface (SPI) slave on the JN517x microcontroller.

Note 1: For information on the SPI slave and guidance on using the SPI Slave functions in JN517x application code, refer to Chapter 15.

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

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

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

bAHI_SpiSlaveEnable

```
bool_t bAHI_SpiSlaveEnable(
                    bool_t  bLsbFirst,
                    uint8   *pu8TxBuffer,
                    uint8   u8TxBufferLength,
                    uint8   u8TxBufferThreshold,
                    uint8   *pu8RxBuffer,
                    uint8   u8RxBufferLength,
                    uint8   u8RxBufferThreshold,
                    uint16  u16RxTimeOut,
                    uint16  u16InterruptEnableMask);
```

Description

This function initialises and configures the SPI Slave.

The function does not, however, configure the pin locations of the SPI slave signals (the JN516x version of this function does). The location of the signals SPI_S_MOSI SPI_S_MISO, SPI_S_SCK and SPI_S_SEL are controlled by calling **vAHI_SetDOpinMultiplexValue()**. The table below shows where the various signals for the SPI slave can be routed:

Signal	Pins
SPI_S_SCK	DIO6, DIO18
SPI_S_MOSI	DIO11, DIO14
SPI_S_MISO	DO1, DIO17
SPI_S_SEL	DIO7, DIO15

Parameters

bLsbFirst	Configures serial bit-order: TRUE - SPI data byte transferred LSB first FALSE - SPI data byte transferred MSB first
**pu8TxBuffer*	Pointer to start of Transmit buffer in RAM
u8TxBufferLength	Length of Transmit buffer, in bytes (1 to 255)
u8TxBufferThreshold	Fill threshold of Transmit buffer, in bytes (0 to 255)
**pu8RxBuffer*	Pointer to start of Receive buffer in RAM
u8RxBufferLength	Length of Receive buffer, in bytes (1 to 255)
u8RxBufferThreshold	Fill threshold of Receive buffer, in bytes (0 to 255)
u16RxTimeOut	Receive timeout duration, in microseconds (0 to 4095)

u16InterruptEnableMask	Interrupt enable mask (bit): †	
	E_AHI_SPIS_INT_RX_FIRST_MASK	(0)
	E_AHI_SPIS_INT_TX_LAST_MASK	(1)
	E_AHI_SPIS_INT_RX_CLIMB_MASK	(2)
	E_AHI_SPIS_INT_TX_FALL_MASK	(3)
	E_AHI_SPIS_INT_RX_OVER_MASK	(4)
	E_AHI_SPIS_INT_TX_OVER_MASK	(5)
	E_AHI_SPIS_INT_RX_UNDER_MASK	(6)
	E_AHI_SPIS_INT_TX_UNDER_MASK	(7)
	E_AHI_SPIS_INT_RX_TIMEOUT_MASK	(8)

† Refer to Table 20 in Appendix B.2 for a description of each mask bit enumeration.

Returns

TRUE if successfully configured, FALSE otherwise (i.e. invalid input parameters)

vAHI_SpiSlaveDisable

```
void vAHI_SpiSlaveDisable(void);
```

Description

This function can be used to disable the SPI Slave.

Parameters

None

Returns

None

vAHI_SpiSlaveReset

```
void vAHI_SpiSlaveReset(bool_t bTxReset,
                        bool_t bRxReset);
```

Description

This function can be used to reset the Transmit and/or Receive FIFO buffers. Following a reset, the internal buffer pointers are re-initialised, the fill-level is reset to zero and the buffer contents remain unchanged.

Parameters

bTxReset	Transmit buffer reset: TRUE - Reset buffer FALSE - Do not reset buffer
bRxReset	Receive buffer reset: TRUE - Reset buffer FALSE - Do not reset buffer

Returns

None

vAHI_SpiSlaveTxWriteByte

void vAHI_SpiSlaveTxWriteByte(uint8 *u8Byte***);**

Description

This function writes a byte of data to the Transmit FIFO buffer of the SPI Slave.

Parameters

u8Byte	Data byte to write to the Transmit FIFO buffer

Returns

None

u8AHI_SpiSlaveRxReadByte

```
uint8 u8AHI_SpiSlaveRxReadByte(void);
```

Description

This function reads a byte of data from the Receive FIFO of the SPI Slave.

Parameters

None

Returns

Data byte read from the Receive FIFO buffer

u8AHI_SpiSlaveTxFillLevel

```
uint8 u8AHI_SpiSlaveTxFillLevel(void);
```

Description

This function returns the fill-level of the Transmit FIFO buffer of the SPI Slave.

Parameters

None

Returns

Fill-level of Transmit FIFO buffer

u8AHI_SpiSlaveRxFillLevel

```
uint8 u8AHI_SpiSlaveRxFillLevel(void);
```

Description

This function returns the fill-level of the Receive FIFO buffer of the SPI Slave.

Parameters

None

Returns

Fill-level of Receive FIFO buffer

u8AHI_SpiSlaveStatus

```
uint8 u8AHI_SpiSlaveStatus(void);
```

Description

This function returns a bitmap indicating the status of the SPI Slave.

Parameters

None

Returns

SPI Slave status bitmap which can be bitwise ANDed with the following masks:

E_AHI_SPIS_STAT_RX_AVAIL_MASK (0x1)	Receive buffer not empty
E_AHI_SPIS_STAT_TX_PENDING_MASK (0x2)	Transmit buffer not empty
E_AHI_SPIS_STAT_RX_ABOVE_MASK (0x4)	Receive buffer fill-level above threshold
E_AHI_SPIS_STAT_TX_ABOVE_MASK (0x8)	Transmit buffer fill-level above threshold

vAHI_SpiSlaveRegisterCallback

> **void vAHI_SpiSlaveRegisterCallback(
> PR_HWINT_APPCALLBACK** *prSpiCallback***);**

Description

This function registers an application callback that will be called when the SPI Slave interrupt is triggered.

The callback function prototype is:

void vHwDeviceIntCallback(uint32 *u32DeviceId***, uint32** *u32ItemBitmap***);**

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

Interrupt handling and the callback function prototype are described in Appendix A.

Parameters

prSpiCallback	Pointer to callback function to be registered

Returns

None

32. Flash Memory Functions

This chapter describes functions for erasing and programming a sector of a Flash memory device. The Flash memory can be the on-chip device or an external device.

Functions are supplied that can be used to interact with any compatible Flash device (detailed in Section 16.1). They are able to access any sector of Flash memory - the application is stored from the first sector (0) and application data is normally stored in the final sector.

> **Note 1:** To access sectors other than the final sector, you should refer to the data sheet for the Flash device to obtain the necessary sector details. However, be careful not to erase essential data such as application code. The application is stored from Sector 0 of the on-chip Flash memory.
>
> **Note 2:** All Flash addresses specified in these functions are offsets from the start of Flash memory and not absolute addresses.
>
> **Note 3:** For guidance on using the Flash memory functions in JN517x application code, refer to Chapter 16.

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

bAHI_FlashInit

```
bool_t bAHI_FlashInit(
                teFlashChipType flashType,
                tSPIflashFncTable *pCustomFncTable);
```

Description

This function selects the type of Flash memory device to be used.

The Flash memory device can be one of the supported device types or a custom device. In the latter case, a custom table of functions must be supplied for interaction with the device.

For information on the Flash memory devices supported by each JN517x microcontroller, refer to Section 16.1.

Note: If you wish to use both internal (on-chip) and external Flash memory devices, you will need to call **bAHI_FlashInit()** when switching between them.

Parameters

flashType — Type of Flash memory device, one of:

E_FL_CHIP_ATMEL_AT25F512	(Atmel AT25F512)
E_FL_CHIP_ST_M25P05_A	(ST M25P05A)
E_FL_CHIP_ST_M25P10_A	(ST M25P10A)
E_FL_CHIP_ST_M25P20_A	(ST M25P20 / Winbond W25X20B) †
E_FL_CHIP_ST_M25P40_A	(ST M25P40)
E_FL_CHIP_SST_25VF010	(Microchip SST25VF010A) ††
E_FL_CHIP_CUSTOM	(custom external device)
E_FL_CHIP_INTERNAL	(on-chip Flash memory)
E_FL_CHIP_AUTO	(on-chip Flash memory)

**pCustomFncTable* — Pointer to the function table for a custom Flash device (E_FL_CHIP_CUSTOM) - see below. If a supported Flash device is used, set to NULL.

† The Winbond W25X20B device is similar to the ST M25P20 device and should be specified as the latter (E_FL_CHIP_ST_M25P20_A).

†† The Microchip SST25VF010A device is supported using 4x32KB overlay blocks instead of 32x4KB sectors.

Custom Flash Functions

If a custom Flash memory device is to be used, a set of custom Flash functions (that will be invoked by the supplied Flash functions to access the custom device) must be provided via the following structure pointed to by the *pCustomFncTable* parameter:

```
typedef struct tagSPIflashFncTable {
    uint32                      u32Signature;
    uint16                      u16FlashId;
    uint16                      u16Reserved;
    tpfvZSPIflashInit           vZSPIflashInit;
    tpfvZSPIflashSetSlaveSel    vZSPIflashSetSlaveSel;
    tpfvZSPIflashWREN           vZSPIflashWREN;
    tpfvZSPIflashEWRSR          vZSPIflashEWRSR;
    tpfu8ZSPIflashRDSR          u8ZSPIflashRDSR;
    tpfu16ZSPIflashRDID         u16ZSPIflashRDID;
    tpfvZSPIflashWRSR           vZSPIflashWRSR;
    tpfvZSPIflashPP             vZSPIflashPP;
    tpfvZSPIflashRead           vZSPIflashRead;
    tpfvZSPIflashBE             vZSPIflashBE;
    tpfvZSPIflashSE             vZSPIflashSE;
} tSPIflashFncTable;
```

The custom function protypes are listed below:

```
typedef void    (*tpfvZSPIflashInit)(int iDivisor, uint8 u8SlaveSel);
typedef void    (*tpfvZSPIflashSetSlaveSel)(uint8 u8SlaveSel);
typedef void    (*tpfvZSPIflashWREN)(void);
typedef void    (*tpfvZSPIflashEWRSR)(void);
typedef uint8   (*tpfu8ZSPIflashRDSR)(void);
typedef uint16  (*tpfu16ZSPIflashRDID)(void);
typedef void    (*tpfvZSPIflashWRSR)(uint8 u8Data);
typedef void    (*tpfvZSPIflashPP)(uint32 u32Addr, uint16 u16Len,
                 uint8* pu8Data);
typedef void    (*tpfvZSPIflashRead)(uint32 u32Addr, uint16 u16Len,
                 uint8* pu8Data);
typedef void    (*tpfvZSPIflashBE)(void);
typedef void    (*tpfvZSPIflashSE)(uint8 u8Sector);
```

Returns

TRUE if initialisation was successful

FALSE if failed

bAHI_FlashEraseSector

```
bool_t bAHI_FlashEraseSector(uint8 u8Sector);
```

Description

This function erases the specified sector of Flash memory by setting all bits to 1.

The function can be used with any compatible Flash memory device with up to 8 sectors. Refer to the data sheet of the Flash memory device for details of its sectors.

> ***Caution:*** *Be careful not to erase essential data such as application code. The application is stored from the start of the on-chip Flash memory (starting in Sector 0).*

Parameters

u8Sector	Number of the sector to be erased (in the range 0 to 7)

Returns

TRUE if sector erase was successful

FALSE if erase failed

bAHI_FlashInternalMultiSectorErase

```
bool_t bAHI_FlashInternalMultiSectorErase(
                                uint8 u8SectorBitmap);
```

Description

This function erases several sectors of the internal Flash memory, as specified by the bitmap parameter.

Parameters

u8SectorBitmap	8-bit bitmap specifying which sectors to erase - a bit set to 1 indicates that the corresponding sector is to be erased

Returns

TRUE if the operation was successful

FALSE otherwise

bAHI_FullFlashProgram

```
bool_t bAHI_FullFlashProgram(uint32 u32Addr,
                             uint16 u16Len,
                             uint8 *pu8Data);
```

Description

This function programs a block of Flash memory by clearing the appropriate bits from 1 to 0. The function can be used to access any sector of a compatible Flash memory device. This function must only be used to write a block of data containing a multiple of 16 bytes and this block must be written to a 16-byte boundary.

This mechanism does not allow bits to be set from 0 to 1. It is only possible to set bits to 1 by erasing the entire sector - therefore, before using this function, you must call the function **bAHI_FlashEraseSector()**.

> ***Caution:*** *Each sector of the internal Flash memory in the JN517x device is divided into 16-byte pagewords. A write to a non-blank pageword must not be performed - the sector containing the non-blank pageword should first be erased using* ***bAHI_FlashEraseSector()*** *before writing to the pageword. If the user omits the sector-erase operation, a subsequent error will likely result when reading from the pageword - this read-error will trigger an interrupt and execute the callback function registered using* ***bAHI_FlashEECerrorInterruptSet()****.*

> ***Caution:*** *The internal Flash memory of the JN517x device has an endurance limit of 10000 write/erase cycles per sector. Refer to the device-specific data sheet for the endurance limit of the external Flash memory.*

Parameters

u32Addr	Address offset from start of Flash memory of first byte to be programmed (must be on a 16-byte boundary)
u16Len	Number of bytes to be programmed (must be a multiple of 16 up to 0x8000)
**pu8Data*	Pointer to start of data block to be written to Flash memory

Returns

TRUE if write was successful

FALSE if write failed

bAHI_FullFlashRead

```
bool_t bAHI_FullFlashRead(uint32 u32Addr,
                          uint16 u16Len,
                          uint8 *pu8Data);
```

Description

This function reads data from the application data area of Flash memory. The function can be used to access any sector of a compatible Flash memory device.

If the function parameters are invalid (e.g. by trying to read beyond end of sector), the function returns without reading anything.

Parameters

u32Addr	Address offset from start of Flash memory of first byte to be read
u16Len	Number of bytes to be read: integer in range 1 to 0x8000
**pu8Data*	Pointer to start of buffer to receive read data.

Returns

TRUE (always)

vAHI_FlashPowerDown

```
void vAHI_FlashPowerDown(void);
```

Description

This function sends a ‘power down’ command to an external Flash memory device attached to the JN517x device. This allows further power savings to be made when the microcontroller is put into a sleep mode (including Deep Sleep).

The following Flash devices are supported by this function:

- STM25P05A
- STM25P10A
- STM25P20
- STM25P40

If the function is called for an unsupported Flash device, the function will return without doing anything.

The application on a JN517x device is responsible for managing the power to external Flash memory for all sleep modes (including Deep Sleep). If the external Flash device is to be unpowered while the JN517x device is sleeping, this function must be called before **vAHI_Sleep()** is called to put the CPU into Sleep mode. You must subsequently power up the device using **vAHI_FlashPowerUp()** after waking and before attempting to access the Flash memory.

> ***Caution:*** *This function must not be called when using the JN517x on-chip Flash memory device - that is, when **bAHI_FlashInit()** has been called with the Flash device type E_FL_CHIP_INTERNAL or E_FL_CHIP_AUTO specified. Note that when using the Persistent Data Manager (PDM), the E_FL_CHIP_AUTO option is used by default, in which case the on-chip Flash memory device will be detected.*

Parameters

None

Returns

None

vAHI_FlashPowerUp

```
void vAHI_FlashPowerUp(void);
```

Description

This function sends a ‘power up’ command to an external Flash memory device attached to the JN517x device.

The following Flash devices are supported by this function:

- STM25P05A
- STM25P10A
- STM25P20
- STM25P40

If the function is called for an unsupported Flash device, the function will return without doing anything.

The application on a JN517x device is responsible for managing the power to external Flash memory for all sleep modes (including Deep Sleep). This function must be called when the JN517x device wakes from sleep if the Flash device was powered down using **vAHI_FlashPowerDown()** before the device entered Sleep mode.

> ***Caution:*** *This function must not be called when using the JN517x on-chip Flash memory device - that is, when **bAHI_FlashInit()** has been called with the Flash device type E_FL_CHIP_INTERNAL or E_FL_CHIP_AUTO specified. Note that when using the Persistent Data Manager (PDM), the E_FL_CHIP_AUTO option is used by default, in which case the on-chip Flash memory device will be detected.*

Parameters

None

Returns

None

bAHI_FlashEECerrorInterruptSet

```
bool_t bAHI_FlashEECerrorInterruptSet(
        bool_t bEnable,
        PR_HWINT_APPCALLBACK prFlashEECCallback);
```

Description

This function can be used to enable or disable interrupts that are generated when an error occurs in the on-chip Flash memory device. A user-defined callback function must be specified which will be invoked when an interrupt of this type occurs.

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

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

Interrupt handling is described in Appendix A.

Parameters

bEnable	Enable or disable internal Flash memory interrupts: TRUE - enable FALSE - disable
prFlashEECCallback	Pointer to callback function to be registered

Returns

None

vAHI_ExtendedTemperatureOperation

void vAHI_ExtendedTemperatureOperation(bool_t *bEnable*);

Description

This function can be used to enable/disable a 'high temperature' feature which is used by some Flash functions. For systems that may operate above the standard temperature limit of 85^{o}C, this feature ensures that the required voltages are applied to the Flash device for correct programming/erasing at these high temperatures. If required, the function should be called soon after a cold-start or warm-start.

Parameters

bEnable	Enable or disable 'high temperature' feature: TRUE - enable FALSE - disable

Returns

None

vAHI_FlashAndEEPROMControllerIntHandler

```
void vAHI_FlashAndEEPROMControllerIntHandler(void);
```

Description

This interrupt handler is called when a parity error interrupt occurs in the Flash controller. It is used to keep track of the number of parity errors recorded and also to call **prFlashEECAppCallback**, if this callback is registered.

Parameters

None

Returns

None

33. EEPROM Functions

This chapter describes functions for accessing the EEPROM device on the JN517x microcontroller - that is, to read from, write to and erase a segment of EEPROM.

Note 1: Although the functions described in this chapter provide direct access to the EEPROM device, it is recommended that the Persistent Data Manager (PDM) is normally used to access this memory. PDM is part of the JN51xx Core Utilities (JCU), supplied in the JN517x ZigBee SDKs, and is described in the *JN51xx Core Utilities User Guide (JN-UG-3116)*.

Note 2: For guidance on using the EEPROM functions in JN517x application code, refer to Chapter 17.

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

u16AHI_InitialiseEEP

uint16 u16AHI_InitialiseEEP(uint8 ****pu8SegmentDatalength*);**

Description

This function initialises the EEPROM for access and returns the following values:

- The number of bytes in each memory segment is returned in the location pointed to by *pu8SegmentDatalength*
- The number of available memory segments in the device is the return value of the function

Note: The final segment of EEPROM is reserved for production data and cannot be written to or erased.

Parameters

******pu8SegmentDatalength*	Pointer to a location to receive the number of bytes per segment

Returns

Number of available memory segments in the device

iAHI_WriteDataIntoEEPROMsegment

```
int iAHI_WriteDataIntoEEPROMsegment(
                    uint16 u16SegmentIndex,
                    uint8 u8SegmentByteAddress,
                    uint8 *pu8DataBuffer,
                    uint8  u8Datalength);
```

Description

This function can be used to write a block of data into a segment of EEPROM.

The data block can be written starting at any point (byte address) within the segment. The data length must not be greater than the amount of memory space up to the end of the segment. The function will not allow an attempt to write data beyond the end of the segment (an overflow) and will return a 'failure' status.

Parameters

u16SegmentIndex	Index of EEPROM segment to be written to (segments are numbered from zero)
u8SegmentByteAddress	Byte address within the segment of the start location for writing data (offset from beginning of segment)
***pu8DataBuffer*	Pointer to start of data block in RAM to be written to EEPROM
u8Datalength	Length of data block to be written, in bytes

Returns

0 - Success

1 - Failure (parameter values were out of range)

iAHI_ReadDataFromEEPROMsegment

```
int iAHI_ReadDataFromEEPROMsegment(
                        uint16 u16SegmentIndex,
                        uint8 u8SegmentByteAddress,
                        uint8 *pu8DataBuffer,
                        uint8 u8Datalength);
```

Description

This function can be used to read a block of data from a segment of EEPROM.

The data block to be read can start at any point (byte address) within the segment. The length of the data block must be specified and must not be greater than the amount of memory space up to the end of the segment. The function will not allow an attempt to read data beyond the end of the segment and will return a 'failure' status.

Parameters

u16SegmentIndex	Index of EEPROM segment to be read (segments are numbered from zero)
u8SegmentByteAddress	Byte address within the segment of the start location for reading data (offset from beginning of segment)
******pu8DataBuffer*	Pointer to start location in RAM where the read data is to be written
u8Datalength	Length of data block to be read, in bytes

Returns

0 - Success

1 - Failure (parameter values were out of range)

iAHI_EraseEEPROMsegment

int iAHI_EraseEEPROMsegment(uint16 *u16SegmentIndex***);**

Description

This function can be used to erase the specified segment of EEPROM.

Parameters

u16SegmentIndex	Index of segment to erase (segments are numbered from zero)

Returns

0 - Success

1 - Failure (parameter values were out of range)

Part III: Appendices

A. Interrupt Handling

Interrupts from the on-chip peripherals are handled by a set of peripheral-specific callback functions. These user-defined functions can be introduced using the appropriate callback registration functions of the Integrated Peripherals API. For example, you can write your own interrupt handler for UART0 and then register this callback function using the **vAHI_Uart0RegisterCallback()** function. The full list of peripheral interrupt sources and the corresponding callback registration functions is provided in the table below.

Interrupt Source	Callback Registration Function
System Controller *	**vAHI_SysCtrlRegisterCallback()**
Analogue Peripherals (ADC)	**vAHI_APRegisterCallback()**
UART 0	**vAHI_Uart0RegisterCallback()**
UART 1	**vAHI_Uart1RegisterCallback()**
Timer 0	**vAHI_Timer0RegisterCallback()**
Timer 1	**vAHI_Timer1RegisterCallback()**
Timer 2 (PWM1)	**vAHI_Timer2RegisterCallback()**
Timer 3 (PWM2)	**vAHI_Timer3RegisterCallback()**
Timer 4 (PWM3)	**vAHI_Timer4RegisterCallback()**
Timer 5 (PWM4)	**vAHI_Timer5RegisterCallback()**
Timer 6 (PWM5)	**vAHI_Timer6RegisterCallback()**
Timer 7 (PWM6)	**vAHI_Timer7RegisterCallback()**
Timer 8 (Analogue Peripheral Timer)	**vAHI_Timer8RegisterCallback()**
Tick Timer	**vAHI_TickTimerRegisterCallback()**
I^2C Interface (2-wire)	**vAHI_I2CRegisterCallback() ****
SPI Master	**vAHI_SpiRegisterCallback()**
SPI Slave	**vAHI_SpiSlaveRegisterCallback()**
Internal Flash Memory	**bAHI_FlashEECerrorInterruptSet()**
Infra-Red Transmitter	**vAHI_InfraredRegisterCallback()**
Encryption Engine	*Refer to AES Coprocessor API Reference Manual (JN-RM-2013)*

Table 11: Interrupt Sources and Callback Registration Functions

* Includes DIO, comparator, wake timer, pulse counter, random number and brownout interrupts

** Used for both I^2C master and I^2C slave interrupts

Note 1: A callback function is executed in interrupt context. You must therefore ensure that the function returns to the main program in a timely manner.

Note 2: The priorities of interrupts from the various interrupt sources can be set using the function **vAHI_InterruptSetPriority()**.

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 11. Enumerations for these sources are provided in the API and are detailed in Appendix B.1.
- *u32ItemBitmap* is a bitmap that identifies the specific cause of the interrupt within the peripheral block identified through *u32DeviceId* above. Masks are provided in the API that allow particular interrupt causes to be checked for. The UART interrupts are an exception as, in their case, an enumerated value is passed via this parameter instead of a bitmap. The masks and enumerations are detailed in Appendix B.2.

A.2 Callback Behaviour

Before invoking one of the callback functions, the API clears the source of the interrupt, so that there is no danger of the same interrupt causing the processor to enter a state of permanently trying to handle the same interrupt (due to a poorly written callback function). This also means that it is possible to have a NULL callback function.

The UARTs are the exception to this rule. When generating a 'receive data available' or 'time-out indication' interrupt, the UARTs will only clear the interrupt once the data has been read from the UART receive buffer. It is therefore vital that if UART interrupts are to be enabled, the callback function handles the 'receive data available' and 'time-out indication' interrupts by reading the data from the UART before returning.

> **Note:** If the Application Queue API is being used, the above issue with the UART interrupts is handled by this API, so the application does not need to deal with it. For more information on this API, refer to the *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*.

A.3 Handling Wake Interrupts

A JN517x microcontroller can be woken from sleep by any of the following sources:

- Wake timer
- DIO
- Comparator
- Pulse counter

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 bitwise ANDing it with masks provided in the API and detailed in Appendix A.1.

> **Note 1:** 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.
>
> **Note 2:** If using the JenNet protocol, do not call these functions to obtain the interrupt status on waking from sleep. At wake-up, JenNet calls **u32AHI_Init()** internally and clears the interrupt status before passing control to the application. The System Controller callback function must be used to obtain the interrupt status, if required.

The above wake sources are outlined below.

Wake Timer

There are two wake timers (0 and 1) on the JN517x microcontroller. These timers run at a nominal 32kHz 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 20 DIO lines (0-19) on the JN517x microcontroller. A JN517x 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 is one comparator (numbered 1) on the JN517x microcontroller. A JN517x 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 comparator 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

There are two pulse counters (0 and 1) on the JN517x microcontroller. These counters are able to run during sleep periods. When a running pulse counter reaches its reference count during sleep, an interrupt can be generated which wakes the device. Control of the pulse counters is described in Chapter 11.

Interrupts for a pulse counter can be enabled when the pulse counter is configured using the function **bAHI_PulseCounterConfigure()**.

B. Interrupt Enumerations and Masks

This appendix details the enumerations and masks used in the parameters of the interrupt callback function described in Appendix A.1.

B.1 Peripheral Interrupt Enumerations (u32DeviceId)

The device ID, *u32DeviceId*, is an enumerated value indicating the peripheral that generated the interrupt. The enumerations are detailed in Table 12 below.

Enumeration	Interrupt Source	Callback Registration Function
E_AHI_DEVICE_SYSCTRL	System Controller	**vAHI_SysCtrlRegisterCallback()**
E_AHI_DEVICE_ANALOGUE	Analogue Peripherals	**vAHI_APRegisterCallback()**
E_AHI_DEVICE_UART0	UART 0	**vAHI_Uart0RegisterCallback()**
E_AHI_DEVICE_UART1	UART 1	**vAHI_Uart1RegisterCallback()**
E_AHI_DEVICE_TIMER0	Timer 0	**vAHI_Timer0RegisterCallback()**
E_AHI_DEVICE_TIMER1	Timer 1	**vAHI_Timer1RegisterCallback()**
E_AHI_DEVICE_TIMER2	Timer 2 (PWM1)	**vAHI_Timer2RegisterCallback()**
E_AHI_DEVICE_TIMER3	Timer 3 (PWM2)	**vAHI_Timer3RegisterCallback()**
E_AHI_DEVICE_TIMER4	Timer 4 (PWM3)	**vAHI_Timer4RegisterCallback()**
E_AHI_DEVICE_TIMER5	Timer 5 (PWM4)	**vAHI_Timer5RegisterCallback()**
E_AHI_DEVICE_TIMER6	Timer 6 (PWM5)	**vAHI_Timer6RegisterCallback()**
E_AHI_DEVICE_TIMER7	Timer 7 (PWM6)	**vAHI_Timer7RegisterCallback()**
E_AHI_DEVICE_TIMER8	Timer 8 (Analogue Peripheral Timer)	**vAHI_Timer8RegisterCallback()**
E_AHI_DEVICE_TICK_TIMER	Tick Timer	**vAHI_TickTimerRegisterCallback()** **vAHI_TickTimerInit()**
E_AHI_DEVICE_I2C *	I^2C Interface (2-wire)	**vAHI_I2CRegisterCallback() ***
E_AHI_DEVICE_SPIM	SPI Master	**vAHI_SpiRegisterCallback()**
E_AHI_DEVICE_SPIS	SPI Slave	**vAHI_SpiSlaveRegisterCallback()**
E_AHI_DEVICE_FEC	Internal Flash Memory	**bAHI_FlashEECerrorInterruptSet()**
E_AHI_DEVICE_INFRARED	Infra-Red Transmitter	**vAHI_InfraredRegisterCallback()**
E_AHI_DEVICE_AES	Encryption Engine	Refer to *AES Coprocessor API Reference Manual (JN-RM-2013)*

Table 12: u32DeviceId Enumerations

* Used for both I^2C master and I^2C slave interrupts

B.2 Peripheral Interrupt Sources (u32ItemBitmap)

The parameter *u32ItemBitmap* is a 32-bit bitmask indicating the individual interrupt source within the peripheral (except for the UARTs, for which the parameter returns an enumerated value). The bits and their meanings are detailed in the tables below.

Mask (Bit)	Description
E_AHI_SYSCTRL_CKEM_MASK (31)	System clock source has been changed
E_AHI_SYSCTRL_RNDEM_MASK (30)	A new value has been generated by the Random Number Generator
E_AHI_SYSCTRL_COMP1_MASK (29) E_AHI_SYSCTRL_COMP0_MASK (28)	Comparator (0 and 1) events
E_AHI_SYSCTRL_WK1_MASK (27) E_AHI_SYSCTRL_WK0_MASK (26)	Wake Timer events
E_AHI_SYSCTRL_VREM_MASK (25) E_AHI_SYSCTRL_VFEM_MASK (24)	Brownout condition entered Brownout condition exited
E_AHI_SYSCTRL_PC1_MASK (23) E_AHI_SYSCTRL_PC0_MASK (22)	Pulse Counter (0 or 1) has reached its pre-configured reference value
E_AHI_DIO18_INT (18) E_AHI_DIO17_INT (17) E_AHI_DIO15_INT (15) . . . E_AHI_DIO0_INT (0)	Digital IO (DIO) events

Table 13: System Controller

Mask (Bit)	Description
E_AHI_AP_INT_DMA_OVER_MASK (4) E_AHI_AP_INT_DMA_END_MASK (3) E_AHI_AP_INT_DMA_MID_MASK (2)	ADC DMA buffer overflow ADC DMA buffer full ADC DMA buffer half-full
E_AHI_AP_ACC_INT_STATUS_MASK (1)	Asserted in ADC accumulation mode to indicate that conversion is complete and the accumulated sample is available
E_AHI_AP_CAPT_INT_STATUS_MASK (0)	Asserted in all ADC modes to indicate that an individual conversion is complete and the resulting sample is available

Table 14: Analogue Peripherals

Mask (Bit)	Description
E_AHI_TIMER_RISE_MASK (1)	Interrupt status, generated on timer rising edge (low-to-high transition) - will be non-zero if interrupt for timer rising output has been set
E_AHI_TIMER_PERIOD_MASK (0)	Interrupt status, generated on end of timer period (high-to-low transition) - will be non-zero if interrupt for end of timer period has been set

Table 15: Timers (identical for all timers)

Mask (Bit)	Description
0	Single source for Tick-timer interrupt, therefore returns 1 every time

Table 16: Tick Timer

Mask (Bit)	Description
E_AHI_INFRARED_TX_MASK (0)	Asserted to indicate transmission complete

Table 17: Infra-Red Transmitter

Mask (Bit)	Description
E_AHI_I2C_MTD_MASK (0x0001)	**Master only.** Master Transaction Done (MTD) - asserted if the Master has completed a transaction successfully
E_AHI_I2C_TAF_MASK (0x0002)	Transmitter Arbitration Failure (TAF) - asserted if the transmitter (master or slave) has lost arbitration
E_AHI_I2C_MTNA_MASK (0x0004)	**Master only.** Master Transmitter No Acknowledge (MTNA) - asserted if the master transmitter has received no acknowledgement from the addressed slave.
E_AHI_I2C_STSD_MASK (0x0008)	**Slave only.** Slave Tx Stop Detect (STSD) - asserted if slave transmitter has detected a stop or restart condition.
E_AHI_I2C_IBE_MASK (0x0040)	I^2C Error (IBE) - asserted if an error has occurred on the I^2C bus.
E_AHI_I2C_MTDR_MASK (0x0080)	**Master only.** Master Transmitter Data Request (MTDR) - asserted if transmission has started and Tx FIFO needs data.
E_AHI_I2C_STDR_MASK (0x0100)	**Slave only.** Slave Transmitter Data Request (STDR) - asserted if transmission has started and Tx FIFO needs data.

Table 18: I^2C Interface

Mask (Bit)	Description
E_AHI_I2C_RFF_MASK (0x0200)	Receive FIFO Full (RFF) - asserted if Rx FIFO is full.
E_AHI_I2C_RFDA_MASK (0x0400)	Receive FIFO Data Available (RFDA) - asserted if Rx FIFO has received data.
E_AHI_I2C_SRSD_RFE_MASK (0x2000)	**Slave only.** Slave Rx Stop Detect and Rx FIFO Empty (SRSD_RFE) - asserted if: • Rx FIFO becomes empty after the slave receiver has detected a stop or restart condition. • Rx FIFO is already empty when the stop or restart condition is detected.

Table 18: I²C Interface

Mask (Bit)	Description
E_AHI_SPIM_TX_MASK (0)	Transfer has completed

Table 19: SPI Master

Mask (Bit)	Description
E_AHI_SPIS_INT_RX_FIRST_MASK (0)	Data has been received in the Receive FIFO, which was previously empty
E_AHI_SPIS_INT_TX_LAST_MASK (1)	Last remaining byte in Transmit FIFO has been transmitted, leaving the buffer empty
E_AHI_SPIS_INT_RX_CLIMB_MASK (2)	Fill-level of Receive FIFO has risen beyond the configured threshold level
E_AHI_SPIS_INT_TX_FALL_MASK (3)	Fill-level of Transmit FIFO has fallen below the configured threshold level
E_AHI_SPIS_INT_RX_OVER_MASK (4)	Data was received but Receive FIFO was full or busy (so data was discarded)
E_AHI_SPIS_INT_TX_OVER_MASK (5)	Transmit FIFO was written to but was full
E_AHI_SPIS_INT_RX_UNDER_MASK (6)	Receive FIFO was read but was empty
E_AHI_SPIS_INT_TX_UNDER_MASK (7)	Transmission was attempted but Transmit FIFO was empty or not ready (so 0x00 was transmitted over the SPI bus)
E_AHI_SPIS_INT_RX_TIMEOUT_MASK (8)	A Receive timeout has occurred (no further data has been received within this period)

Table 20: SPI Slave

For the UART interrupts, *u32ItemBitmap* returns the following enumerated values:

Enumerated Value	Description (and Priority)
E_AHI_UART_INT_RXLINE (3)	Receive line status (highest priority)
E_AHI_UART_INT_RXDATA (2)	Receive data available (next highest priority)
E_AHI_UART_INT_TIMEOUT (6)	Time-out indication (next highest priority)
E_AHI_UART_INT_TX (1)	Transmit FIFO empty (next highest priority)
E_AHI_UART_INT_MODEM (0)	Modem status (lowest priority)

Table 21: UART (identical for both UARTs)

Table 21 lists the UART interrupts from highest priority to lowest priority.

Revision History

Version	Date	Comments
1.0	2-Mar-2016	First release
1.1	5-Oct-2016	Manual re-formatted, I^2C Interface description updated, function vAHI_ModuleConfigure() added and other minor corrections made
1.2	22-Dec-2016	Descriptions of functions bAHI_UartEnable() and vAHI_TimerEnable() added. References to obsolete function vAHI_TimerDIOControl() removed

Important Notice

Limited warranty and liability - Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use - NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications - Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control - This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

NXP Semiconductors

For online support resources and contact details of your local NXP office or distributor, refer to:

www.nxp.com