Legacy documentation
refer to the Altium Wiki for current information

WB_JPGDEC JPEG Decoder

Summary

Core Reference
CR0174 (v2.0) March 07, 2008

This document provides detailed reference information with respect to the WB_JPGDEC peripheral device. This device is used to decode grayscale or color baseline JPEG-compressed image data.

The WB_JPGDEC peripheral facilitates the decoding of baseline JPEG-compressed images (grayscale and color) into RGB565 pixel format output that can be written directly to screen display memory, or to a continuous external memory storage area.

The full JPEG image can be decoded, or only a specified area. The peripheral also supports block-based reading and writing.

Features

- Supports decoding of grayscale and color baseline (sequential) JPEG-formatted images
- Output pixel format is Big-Endian, RGB565
- Supports direct decoding to display memory
- Ability to select an area of an image to decode
- Block-based reading or writing – can be interrupt driven
- 32-bit data interface to host processor
- 32-bit DMA (Direct Memory Access) interface
- Wishbone-compliant

Available Devices

From a schematic document, the WB_JPGDEC device can be found in the FPGA Peripherals integrated library (`FPGA Peripherals.IntLib`), located in the `\Library\Fpga` folder of the installation.

From an OpenBus System document, the JPEG Decoder component can be found in the **Peripherals** region of the **OpenBus Palette** panel.

Legacy documentation
refer to the Altium Wiki for current information

Functional Description

Symbol

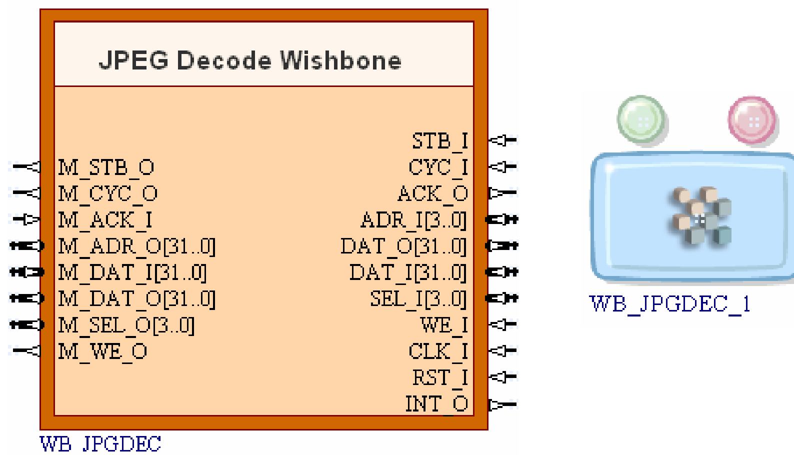

Figure 1. Symbols used for the JPEG Decoder in both schematic (left) and OpenBus System (right).

Pin Description

The following pin description is for the processor when used on the schematic. In an OpenBus System, although the same signals are present, the abstract nature of the system hides the pin-level Wishbone interfaces.

Table 1. WB_JPGDEC pin description

Name	Type	Polarity/ Bus size	Description
			Control Signals
CLK_I	I	Rise	External (system) clock signal
RST_I	I	High	External (system) reset
			Host Processor Interface Signals
STB_I	I	High	Strobe signal. When asserted, indicates the start of a valid Wishbone data transfer cycle
CYC_I	I	High	Cycle signal. When asserted, indicates the start of a valid Wishbone cycle
ACK_O	O	High	Standard Wishbone device acknowledgement signal. When this signal goes high, the WB_JPGDEC (Wishbone Slave) has finished execution of the requested action and the current bus cycle is terminated
ADR_I	I	4	Address bus, used to select an internal register of the device for writing to/reading from
DAT_O	O	32	Data to be sent to host processor
DAT_I	I	32	Data received from host processor
SEL_I	I	4/High	Select input, used to determine where data is placed on the DAT_O line during a Read cycle, and from where on the DAT_I line data is accessed during a Write cycle. Each of the data ports is 32-bits wide with 8-bit granularity, meaning data transfers can be 8-, 16-, or 32-bit. The four select bits allow targeting of each of the four active bytes of a port, with bit 0 corresponding to the low byte (7..0) and bit 3 corresponding to the high byte (31..24)

Legacy documentation
refer to the Altium Wiki for current information

Name	Type	Polarity/ Bus size	Description
WE_I	I	Level	Write enable signal. Used to indicate whether the current local bus cycle is a Read or Write cycle: 0 = Read 1 = Write
INT_O	O	High	Interrupt output line. This output will be taken High if any of the readable bits in the Status register become set (STATUS.8, STATUS6..0) and the corresponding bit in the Interrupt Mask register is also set.
DMA Interface Signals			
M_STB_O	O	High	Strobe signal. When asserted, indicates the start of a valid Wishbone data transfer cycle
M_CYC_O	O	High	Cycle signal. When asserted, indicates the start of a valid Wishbone cycle. This signal remains asserted until the end of the bus cycle, where such a cycle can include multiple data transfers
M_ACK_I	I	High	Standard Wishbone device acknowledgement signal. When this signal goes high, the connected Wishbone memory device has finished execution of the requested action and the current bus cycle is terminated
M_ADR_O	O	32	Standard Wishbone address bus, used to select an address of the connected Wishbone memory for writing to/reading from
M_DAT_I	I	32	Data received from external Wishbone memory
M_DAT_O	O	32	Data to be sent to external Wishbone memory
M_SEL_O	O	4/High	Select output, used to determine where data is placed on the M_DAT_O line during a Write cycle, and from where on the M_DAT_I line data is accessed during a Read cycle. For the WB_JPGDEC, only 32-bit data transfers to/from Wishbone memory are supported, meaning that all the lines go High during a Write or Read cycle
M_WE_O	O	Level	Write enable signal. Used to indicate whether the current local bus cycle is a Read or Write cycle: 0 = Read 1 = Write

Legacy documentation
refer to the Altium Wiki for current information

Hardware Description

Block Diagram

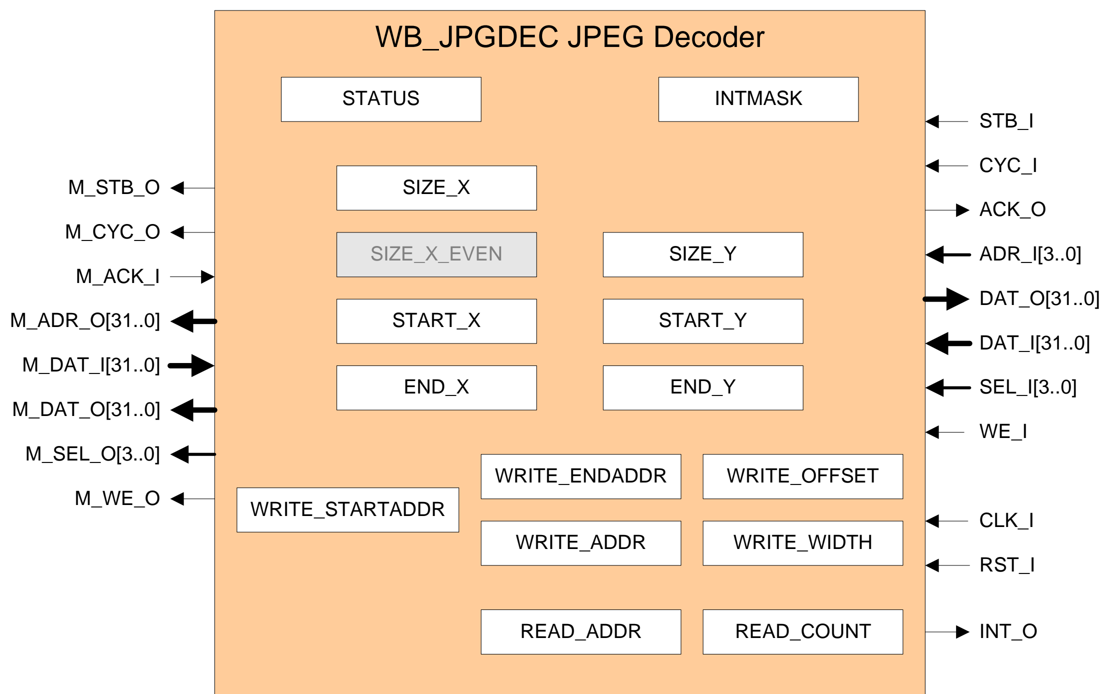

Figure 2. WB_JPGDEC block diagram.

Internal Registers

The following sections detail the internal registers for the WB_JPGDEC that can be accessed from the host processor.

Status Register (STATUS)

Address: 0h

Access: Read only except where indicated

Value after Reset: 0000_0100h

This register reflects the state of the WB_JPGDEC at all times.

Table 2. The STATUS register

MSB LSB

31 9	8	7	6	5	4	3	2	1	0
-	rst	start	unsup	corupt	notjpg	jpgerr	wfull	rempty	jpgrdy

Table 3. The STATUS register bit functions

Bit	Symbol	Function
STATUS.31..STATUS.9	-	Not Used.
STATUS.8 (R/W)	rst	Reset bit. Writing a '1' to this bit issues a software reset – forcing the end of decoding and placing the WB_JPGDEC back into the idle state. As a result of setting this bit, the `jpgerr`, `notjpg`, `corupt`, `unsup`, `wfull` and `rempty` bits will all be cleared. This flag will remain set while the WB_JPGDEC is in the idle state. Cleared by writing '1' to `start` bit.
STATUS.7 (W)	start	Start bit. Writing a '1' to this bit results in a software reset, followed by the start

Legacy documentation
refer to the Altium Wiki for current information

Bit	Symbol	Function
		of decoding. The wfull and rempty bits will both be cleared when this bit is set. Setting this bit will also clear all error flags (jpgerr, notjpg, corupt and unsup).
STATUS.6 (R)	unsup	Unsupported flag. This bit will be set if the format of the supplied JPEG image is not supported (i.e. is not a baseline JPEG). Cleared by writing '1' to rst or start bits.
STATUS.5 (R)	corupt	Corrupt flag. This bit will be set if an unknown error occurs while processing the JPEG image. Cleared by writing '1' to rst or start bits.
STATUS.4 (R)	notjpg	Not a JPEG flag. This bit will be set if the input is not marked as a JPEG image. Cleared by writing '1' to rst or start bits.
STATUS.3 (R)	jpgerr	JPEG Error flag. This bit will be set if any error occurs while processing the JPEG image (i.e. notjpg OR corupt OR unsup become set). Processing itself will be aborted. The notjpg, corupt and unsup flags can be used to gain more information about the nature of the error. Cleared by writing '1' to rst or start bits.
STATUS.2 (R/W)	wfull	Write Destination Full flag. This bit will be set if the output data is about to be written outside of the output address window specified by the values in the WRITE_STARTADDR and WRITE_ENDADDR registers. If set, write to this bit with a '1' to clear it and resume decoding.
STATUS.1 (R/W)	rempty	Read Buffer Empty flag. This bit will be set if the Read Buffer becomes empty and more data is needed. If set, write to this bit with a '1' to clear it and resume decoding.
STATUS.0 (R)	jpgrdy	JPEG Ready Flag. This bit will be set if the entire JPEG image has been correctly processed. Cleared by writing '1' to rst or start bits.

Note: Values written to read-only bits in the register (unsup, corupt, notjpg, jpgerr and jpgrdy) will be ignored. The write-only start bit will return '0' when read.

Interrupt Mask Register (INTMASK)

Address: 1h

Access: Read and Write

Value after Reset: 0000_0000h

This register is used to enable interrupt generation for each of the readable bits in the Status register (STATUS.7, STATUS.6..0). Provided bit INTMASK.n is High, an interrupt will be generated when the corresponding bit STATUS.n goes High.

Table 4. The INTMASK register

MSB LSB

31 … 9	8	7	6	5	4	3	2	1	0
-	rst	-	unsup	corupt	notjpg	jpgerr	wfull	rempty	jpgrdy

Table 5. The INTMASK register bit functions

Bit	Symbol	Function
INTMASK.31..INTMASK.9	-	Not Used.
INTMASK.8	rst	Enables interrupt generation for the Reset flag (STATUS.8).
INTMASK.7	-	Not used. Values written to this bit will be ignored. Returns '0' when read.
INTMASK.6	unsup	Enables interrupt generation for the Unsupported flag (STATUS.6).

Legacy documentation
refer to the Altium Wiki for current information

Bit	Symbol	Function
INTMASK.5	corupt	Enables interrupt generation for the Corrupt flag (STATUS.5).
INTMASK.4	notjpg	Enables interrupt generation for the Not a JPEG flag (STATUS.4).
INTMASK.3	jpgerr	Enables interrupt generation for the JPEG Error flag (STATUS.3).
INTMASK.2	wfull	Enables interrupt generation for the Write Destination Full flag (STATUS.2).
INTMASK.1	rempty	Enables interrupt generation for the Read Buffer Empty flag (STATUS.1).
INTMASK.0	jpgrdy	Enables interrupt generation for the JPEG Ready flag (STATUS.0).

Image Size Registers (SIZE_X, SIZE_Y)

Address: 2h and 3h

Access: Read only (from processor)

Value after Reset: 0000_0000h

These 32-bit registers are used to store 16-bit values for the width (SIZE_X) and height (SIZE_Y) of the image, in pixels, respectively. This information is obtained directly from the JPEG image itself.

The SIZE_X register always stores the actual width of the JPEG image, regardless of whether it is an odd or even number of pixels. However internally, the WB_JPGDEC ensures only even values for width are used. This is achieved using an additional internal register, SIZE_X_EVEN, loaded with bits 15..1 of the SIZE_X register, and '0' into its LSB. This register is not accessible by the processor.

Note: Bits 31..16 of these registers are not used and will return '0' when read.

Table 6. The SIZE_X register

MSB | | LSB

31 … 16	15 … 0
-	size_x

Table 7. The SIZE_X_EVEN register

MSB | | LSB

31 … 16	15 … 1	0
-	size_x_even	0

Table 8. The SIZE_Y register

MSB | | LSB

31 … 16	15 … 0
-	size_y

Decoding Area Registers (START_X, START_Y, END_X, END_Y)

Address: 4h, 5h, 6h, 7h

Access: Read and Write

Value after Reset: 0000_0000h

These 32-bit registers together define the area of the JPEG image that is to be decoded. The 16-bit values in the START_X and START_Y registers define the top-left corner of the decoding area. The 16-bit values in the END_X and END_Y registers define the bottom-right corner of the decoding area.

If a decode area is not explicitly defined, the entire image will be decoded, using the values for the width and height of the JPEG image (stored in the SIZE_X_EVEN and SIZE_Y registers respectively).

Legacy documentation
refer to the Altium Wiki for current information

The top-left corner of the area will be the first pixel written to memory, at the address defined by the initial value stored in the Write Address register (WRITE_ADDR).

As two pixels are written to memory at the same time during decoding, the pixel boundary must be even and so START_X and END_X values must be multiples of 2. This is handled upon writing to these registers by truncating the value and ensuring that '0' is loaded into the LSB.

Note: Bits 31..16 of these registers are not used and will return '0' when read.

Table 9. The START_X register

MSB					LSB
31	16	15		1	0
-		start_x			0

Table 10. The START_Y register

MSB			LSB
31	16	15	0
-		start_y	

Table 11. The END_X register

MSB					LSB
31	16	15		1	0
-		end_x			0

Table 12. The END_Y register

MSB			LSB
31	16	15	0
-		end_y	

Read Address Register (READ_ADDR)

Address: 8h

Access: Read and Write

Value after Reset: XXXX_XXXXh

This register is used to store the 32-bit address in memory from where the WB_JPGDEC will start reading the next data of the JPEG image.

While reading, the value in the READ_ADDR register will be incremented (ready to get the next byte of the image).

Read Count Register (READ_COUNT)

Address: 9h

Access: Read and Write

Value after Reset: 0000_0000h

This register is used to store a 32-bit value for the number of bytes that the WB_JPGDEC will read from the JPEG image, starting from the initial byte at the initial address loaded into the READ_ADDR register.

While reading, the value in the READ_COUNT register will be decremented. If READ_COUNT reaches zero and the JPEG is not yet fully decoded, the `rempty` bit in the Status register (STATUS.1) will be set – flagging that no more data can be read. To resume decoding, the following must be performed:

- Write new values to the READ_ADDR and READ_COUNT registers
- Clear the `rempty` bit in the Status register. This is achieved by writing a '1' to this bit.

Legacy documentation
refer to the Altium Wiki for current information

Write Start Address Register (WRITE_STARTADDR)

Address: Ah

Access: Read and Write

Value after Reset: 0000_0000h

This register is used to define a starting address in memory at which the resulting decoded image can be written. The value in this register defines the lower limit in memory that the WB_JPGDEC will write into, irrespective of the value specified in the WRITE_OFFSET register, or the size of the JPEG.

The address must be located at a 32-bit boundary and therefore the value stored for the start address must be a multiple of 4. Bits 1..0 are therefore always '0'.

Table 13. The WRITE_STARTADDR register

MSB LSB

31 … 2	1	0
write_start_address	0	0

Write End Address Register (WRITE_ENDADDR)

Address: Bh

Access: Read and Write

Value after Reset: 0000_0000h

This register is used to define an end address in memory, up to which the resulting decoded image can be written. The value in this register defines the upper limit in memory that the WB_JPGDEC will write into, irrespective of the value specified in the WRITE_OFFSET register, or the size of the JPEG.

The address must be located at a 32-bit boundary and therefore the value stored for the end address must be a multiple of 4. Bits 1..0 are therefore always '0'.

Table 14. The WRITE_ENDADDR register

MSB LSB

31 … 2	1	0
write_end_address	0	0

Write Offset Register (WRITE_OFFSET)

Address: Ch

Access: Read and Write

Value after Reset: 0000_0000h

This register is used to store an offset value, allowing you to shift where in memory the resulting decoded image will start to be written. The offset is relative to the value stored for the start address, in the WRITE_STARTADDR register.

The address must be located at a 32-bit boundary and therefore the value stored for the offset must be a multiple of 4. Bits 1..0 are therefore always '0'.

Table 15. The WRITE_OFFSET register

MSB LSB

31 … 2	1	0
write_offset	0	0

Legacy documentation
refer to the Altium Wiki for current information

Write Width Register (WRITE_WIDTH)

Address: Dh

Access: Read and Write

Value after Reset: 0000_0000h

This register is used to store a 16-bit value for the required width of the decoded output image, in pixels. If the decoded JPEG image is being written directly to screen, the value for the screen line width, in pixels should be entered into this register. If the decoded image is being written as an image in continuous memory, this register should be left with the value of zero. In the latter case, the value stored for the width of the JPEG image – in the SIZE_X_EVEN register – will be used instead.

How the value in the WRITE_WIDTH register is used, depends on the width of the original JPEG image being decoded:

- If the JPEG is smaller in width than the required width, pixels will be skipped during decoding.
- If the JPEG is greater in width than the required width, pixels will be discarded during decoding.

If the value in the WRITE_WIDTH register is zero, then all pixels of image lines will be written to memory, with no gaps between lines.

As two pixels are written to memory at the same time during decoding, the pixel boundary must be even and so the value in the WRITE_WIDTH register must be a multiple of 2. This is handled upon writing to the register by truncating the value and ensuring that '0' is loaded into the LSB.

Note: Bits 31..16 of this register are not used and will return '0' when read.

Table 16. The WRITE_WIDTH register

MSB LSB

31 16	15 1	0
-	write_width	0

Write Address Register (WRITE_ADDR)

Address: Eh

Access: Read only

Value after Reset: 0000_0000h

This register is used to contain the address in memory at which the WB_JPGDEC will write the next pixel of the decoded image.

The actual address for a write must be located at a 32-bit boundary and therefore the value stored for the address must be a multiple of 4. Bits 1..0 are therefore always '0'.

The actual address of an RGB565 formatted pixel (X, Y) written by the WB_JPGDEC can be summarized by the expression:

```
write_addr = write_startaddr - write_offset + ((X + (write_width * Y)) * 2)
```

where:

- `write_addr` is the value stored in the WRITE_ADDR register
- `write_startaddr` is the value stored in the WRITE_STARTADDR register
- `write_offset` is the value stored in the WRITE_OFFSET register
- `write_width` is the value stored in the WRITE_WIDTH register. Note that if the value in this register is zero, then `size_x_even`, the value in the SIZE_X_EVEN register will be used instead.

If the WB_JPGDEC tries to write a pixel while the value stored in the WRITE_ADDR register is outside of the target memory address window – defined by the values in the WRITE_STARTADDR and WRITE_ENDADDR registers – the `wfull` bit in the Status register (STATUS.2) will be set and decoding of the JPEG image will be suspended. To resume decoding, perform the following:

- Write new values to at least one of the following registers: WRITE_STARTADDR, WRITE_ENDADDR, WRITE_OFFSET
- Clear the `wfull` bit in the Status register. This is achieved by writing a '1' to this bit.

Legacy documentation
refer to the Altium Wiki for current information

Table 17. The WRITE_ADDR register

MSB			LSB
31	2	1	0
write_addr		0	0

Interrupts

The WB_JPGDEC provides for a single external interrupt line to the host processor. This line – INT_O – will be taken High if any of the following readable bits in the Status register become set:

- `jpgrdy` (STATUS.0)
- `rempty` (STATUS.1)
- `wfull` (STATUS.2)
- `jpgerr` (STATUS.3)
- `notjpg` (STATUS.4)
- `corupt` (STATUS.5)
- `unsup` (STATUS.6)
- rst (STATUS.8)

and provided that the corresponding bit in the Interrupt Mask register is also set.

Clearance of an interrupt requires that the Status register flag causing that interrupt be cleared. The following table summarizes how the interrupt state is cleared for each of these bits.

Table 18. Clearing interrupt causes

Interrupt Cause	Action to clear interrupt
`jpgrdy`	This interrupt, if enabled, will be present as long as the WB_JPGDEC is in the ready state – after successful decoding of the JPEG image. It is cleared by writing '1' to either the `rst` or `start` bits in the Status register (STATUS.8 or STATUS.7 respectively).
`rempty`	Write new values to the READ_ADDR and READ_COUNT registers. Then write a '1' to this bit to clear it and resume decoding.
`wfull`	Write new values to at least one of the following registers: WRITE_STARTADDR, WRITE_ENDADDR, WRITE_OFFSET. Then write a '1' to this bit to clear it and resume decoding.

`jpgerr`
`notjpg`
`corupt`
`unsup`
	The `notjpg`, `corupt` and `unsup` flags generate interrupts for specific decoding errors.
	The `jpgerr` bit is purely a global flag for an error having occurred. It will become set, and remain set, while any of the specific error flags remain set.
	Clear all specific (and therefore global) error flags by writing '1' to either the `rst` or `start` bits in the Status register (STATUS.8 or STATUS.7 respectively).

`rst`	This interrupt, if enabled, will be present as long as the WB_JPGDEC is in the idle state. To clear, simply write to the `start` bit in the Status register (STATUS.7) to commence decoding and move out of the idle state.

Legacy documentation
refer to the Altium Wiki for current information

Interfacing to a 32-bit Processor

How the WB_JPGDEC is placed and wired within an FPGA design depends on the method used to build that design. The main processor-based system can be defined purely on the schematic sheet, or it can be contained as a separate OpenBus System, which is then referenced from the top-level schematic. The following sections take a look at using the WB_JPGDEC in both of these design arenas.

Design using a Schematic only

Figure 3 illustrates how a WB_JPGDEC device can be wired into a schematic-based design that uses a 32-bit processor – in this case a TSK3000A. A configurable Wishbone Interconnect device (WB_INTERCON) is used to simplify connection and also handle the address mapping – taking the 24-bit address line from the processor and mapping it to the 4-bit address line used to drive the WB_JPGDEC.

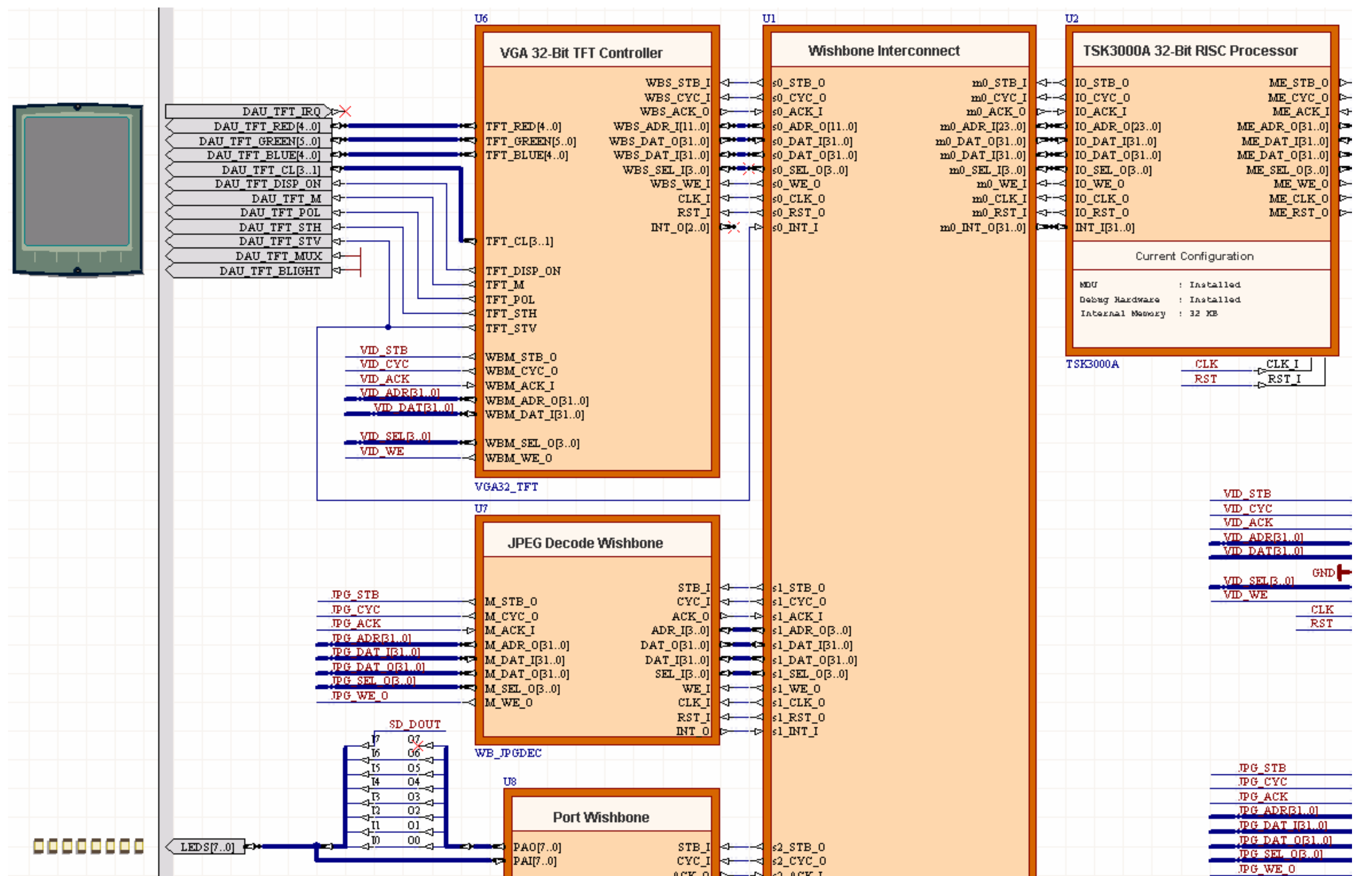

Figure 3. Example interfacing between a 32-bit processor (TSK3000A) and a WB_JPGDEC.

When configuring the WB_INTERCON device – in particular the WB_JPGDEC slave interface – ensure that the Address Bus Mode is set to `Word Addressing - ADR_O(0) <= ADR_I(1 or 2)`. As the WB_JPGDEC's data bus width is 32-bit, the two lowest address bits are not connected to the slave device. ADR_I(2) of the master is mapped to ADR_O(0) of the slave, providing sequential word addresses (or addresses at every 4 bytes). Bits 5..2 of the output address line from the host processor (IO_ADR_O) are therefore mapped, through the WB_INTERCON, to bits 3..0 of the WB_JPGDEC's input address line (ADR_I).

The actual 24-bit address sent from the processor on its IO_ADR_O line is constructed as follows:

WB_JPGDEC Base Address + (Internal Register Address & "00")

The Base Address for the WB_JPGDEC is specified as part of the peripheral's definition when adding it as a slave to the Wishbone Interconnect. For example, if the base address entered for the device is 100000h (mapping it to address FF10_0000h in the processor's address space), and you want to write to the Write Width register (WRITE_WIDTH) with address Dh, the value entered on the processor's IO_ADR_O line would be:

100000h + 34h = 100034h

Legacy documentation
refer to the Altium Wiki for current information

For further information on the Wishbone Interconnect, refer to the *CR0150 WB_INTERCON Configurable Wishbone Interconnect* core reference.

For further information on the TSK3000A processor, refer to the *CR0121 TSK3000A 32-bit RISC Processor* core reference. Similar references can be found for other 32-bit processors supported by Altium Designer, by using the lower section of the **Knowledge Center** panel and navigating to the *Documentation Library » Embedded Processors and Software Development » FPGA Based and Discrete Processors* section.

For an example schematic-based FPGA design featuring a WB_JPGDEC device, refer to the example project: `\Examples\NB2DSK1 Examples\DSF Slideshow\DSF_Slideshow.PrjFpg`.

Design Featuring an OpenBus System

Figure 4 illustrates identical use of the WB_JPGDEC peripheral within a design where the main processor system has been defined as an OpenBus System. The JPEG Decoder peripheral (as it is referred to in the OpenBus System world) is connected to the TSK3000A processor through an Interconnect component.

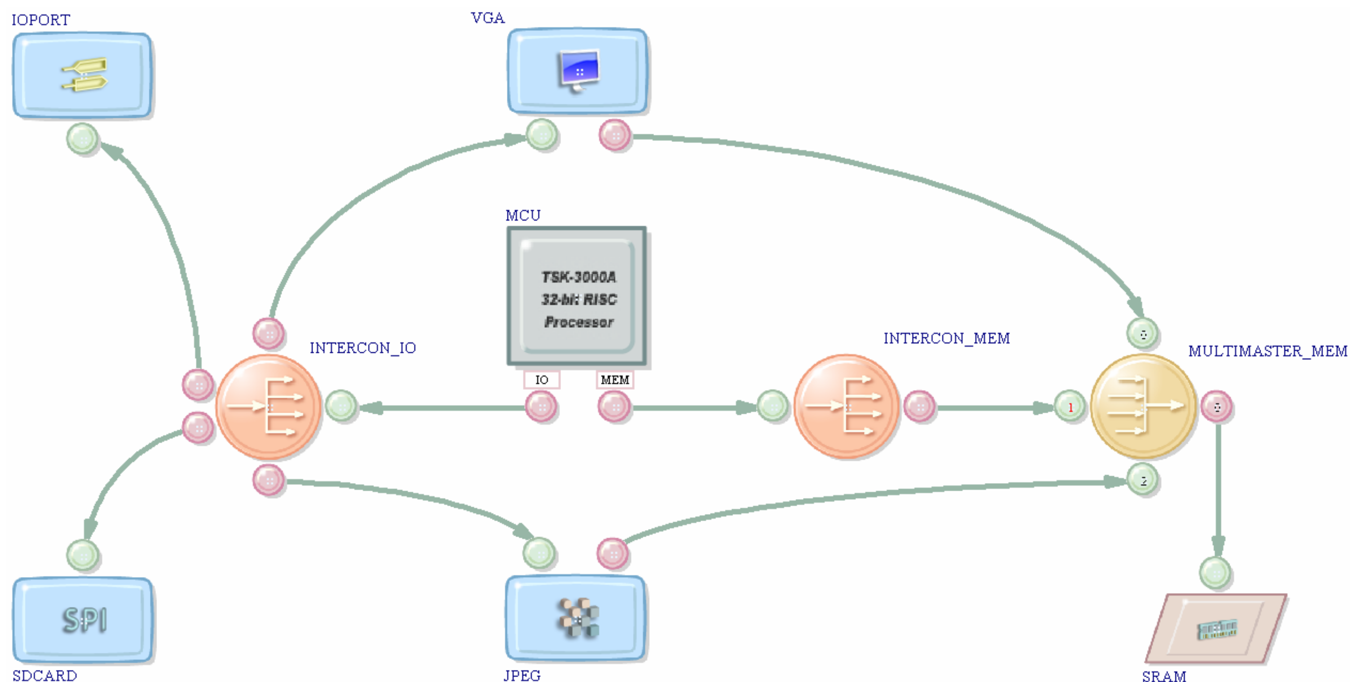

Figure 4. Example interfacing between a 32-bit processor (TSK3000A) and a JPEG Decoder device, as part of an OpenBus System.

The OpenBus System environment is a much more abstract and intuitive place to create a design, where the interfaces are reduced to single ports and connection is made courtesy of single links.

Much of the configuration is handled for you – there is no addressing mode to specify, no data width to enter – the JPEG Decoder peripheral is automatically added as a slave to the Interconnect component by virtue of its link. The Interconnect contains information regarding the device's address bus size and a default decoder address width. All that is really needed is specification of the peripheral's base address – where in the TSK3000A's address space it is to be mapped.

For further information on the Interconnect component, refer to the document *TR0170 OpenBus Interconnect Component Reference*.

For more information on the concepts and workings of the OpenBus System, refer to the article *AR0144 Streamlining Processor-based FPGA design with the OpenBus System*.

For an example OpenBus System-based FPGA design featuring a JPEG Decoder device, refer to the example project: `\Examples\NB2DSK1 Examples\OpenBus Slideshow\DSF_Slideshow.PrjFpg`. This illustrates use of the device to decode a series of JPEG images, stored on an SD card, and show each one on the Desktop NanoBoard's TFT LCD panel.

Legacy documentation
refer to the Altium Wiki for current information

Host to Controller Communications

Communications between a 32-bit host processor and the WB_JPGDEC are carried out over a standard Wishbone bus interface. The following sections detail the communication cycles involved between host and peripheral device for writing to/reading from the internal registers.

Writing to an Internal Register

Data is written from the host processor to an internal register in the WB_JPGDEC, in accordance with the standard Wishbone data transfer handshaking protocol. The write operation occurs on the rising edge of the CLK_I signal and can be summarized as follows:

- The host presents the required 24-bit address based on the register to be written on its IO_ADR_O output and valid data on its IO_DAT_O output. It then asserts its IO_WE_O signal, to specify a write cycle
- The WB_JPGDEC receives the 4-bit address on its ADR_I input and, identifying the addressed register, prepares to receive data into that register
- The host asserts its IO_STB_O and IO_CYC_O outputs, indicating that the transfer is to begin. The WB_JPGDEC, which monitors its STB_I and CYC_I inputs on each rising edge of the CLK_I signal, reacts to this assertion by latching the data appearing at its DAT_I input into the target register and asserting its ACK_O signal – to indicate to the host that the data has been received
- The host, which monitors its IO_ACK_I input on each rising edge of the CLK_I signal, responds by negating the IO_STB_O and IO_CYC_O signals. At the same time, the WB_JPGDEC negates the ACK_O signal and the data transfer cycle is naturally terminated.

Table 19 summarizes how the 32-bit data word from the host processor is used by each of the internal registers.

Table 19. Values written to internal registers during a write.

Writing to...	Results in...
STATUS	If DAT_I(8) = '1': • STATUS(6..1) are cleared to '0' If DAT_I(7) = '1': • STATUS.8 is cleared to '0' • STATUS[6..1] are cleared to '0' If DAT_I(2) = '1': • STATUS.2 is cleared to '0' If DAT_I(1) = '1': • STATUS.1 is cleared to '0'
INTMASK	DAT_I(8..0) loaded into the Interrupt Mask register
START_X	DAT_I(15..1) & "0" loaded into the START_X register
START_Y	DAT_I(15..0) loaded into the START_Y register
END_X	DAT_I(15..1) & "0" loaded into the END_X register
END_Y	DAT_I(15..0) loaded into the END_Y register
READ_ADDR	DAT_I(31..0) loaded into the Read Address register
READ_COUNT	DAT_I(31..0) loaded into the Read Count register
WRITE_STARTADDR	DAT_I(31..2) and "00" loaded into the Write Start Address register
WRITE_ENDADDR	DAT_I(31..2) and "00" loaded into the Write End Address register
WRITE_OFFSET	DAT_I(31..2) and "00" loaded into the Write Offset register
WRITE_WIDTH	DAT_I(15..1) & "0" loaded into the Write Width register

Legacy documentation
refer to the Altium Wiki for current information

Reading from an Internal Register

Data is read from an internal register in accordance with the standard Wishbone data transfer handshaking protocol. The read operation, which occurs on the rising edge of the CLK_I signal, can be summarized as follows:

- The host presents the required 24-bit address based on the register to be read on its IO_ADR_O output. It then negates its IO_WE_O signal, to specify a read cycle
- The WB_JPGDEC receives the 4-bit address on its ADR_I input and, identifying the addressed register, prepares to transmit data from the selected register
- The host asserts its IO_STB_O and IO_CYC_O outputs, indicating that the transfer is to begin. The WB_JPGDEC, which monitors its STB_I and CYC_I inputs on each rising edge of the CLK_I signal, reacts to this assertion by presenting the valid data on its DAT_O output and asserting its ACK_O signal – to indicate to the host that valid data is present
- The host, which monitors its IO_ACK_I input on each rising edge of the CLK_I signal, responds by latching the data appearing at its IO_DAT_I input and negating the IO_STB_O and IO_CYC_O signals. At the same time, the WB_JPGDEC negates the ACK_O signal and the data transfer cycle is naturally terminated.

Table 20 summarizes the 'make-up' of the 32-bit data word that is read back from each register.

Table 20. Values read from internal registers during a read.

Reading from...	Presents (to host processor)...
STATUS	"000000000000000000000000" & 8-bit value from the Status register
INTMASK	"000000000000000000000000" & 8-bit value from the Interrupt Mask register
SIZE_X	"0000000000000000" & 16-bit value from the SIZE_X register
SIZE_Y	"0000000000000000" & 16-bit value from the SIZE_Y register
START_X	"0000000000000000" & 16-bit value from the START_X register
START_Y	"0000000000000000" & 16-bit value from the START_Y register
END_X	"0000000000000000" & 16-bit value from the END_X register
END_Y	"0000000000000000" & 16-bit value from the END_Y register
READ_ADDR	32-bit value from the Read Address register
READ_COUNT	32-bit value from the Read Count register
WRITE_STARTADDR	32-bit value from the Write Start Address register
WRITE_ENDADDR	32-bit value from the Write End Address register
WRITE_OFFSET	32-bit value from the Write Offset register
WRITE_WIDTH	"0000000000000000" & 16-bit value from the Write Width register
WRITE_ADDR	32-bit value from the Write Address register

Legacy documentation
refer to the Altium Wiki for current information

Operational Overview

The following sections take a look at initialization of the WB_JPGDEC and example usage.

Initialization

After an external reset on the RST_I line, you will need to initialize the WB_JPGDEC. This should be carried out in accordance with design requirements and can include:

- Loading the Interrupt Mask register with a value that will enable the required interrupts for the peripheral.
- If you want to decode a specific area of the JPEG, load the START_X, START_Y, END_X and END_Y registers, with the left column, right column, top line and bottom line respectively.
- Setup the input buffer for decoding by loading the starting address for the buffer into the Read Address register and the size of the buffer into the Read Count register.
- Setup the output buffer for decoding by loading the starting address for the buffer into the Write Start Address register, the end address for the buffer into the Write End Address register and the width for an output line (in pixels) into the Write Width register. If you need to offset the location, enter the offset value into the Write Offset register.

Once the registers have been set as required, simply write a '1' to the `start` bit in the Status register (STATUS.7), to begin decoding. Once the image has been successfully decoded, the WB_JPGDEC will remain in the ready state (`jpgrdy` bit set in the Status register). To decode another image, you will again need to write '1' to the `start` bit in the Status register.

Note: If you issue a soft reset, by writing '1' to the `rst` bit of the Status register (STATUS.8), the processor-accessible internal registers themselves are not reset and therefore will retain their previous values. You may, however, need to adjust the values in the Read Address and Read Count registers before restarting the decoder.

Example Usage

To better illustrate the use of the internal registers, the following sections provide examples of how the WB_JPGDEC peripheral can be used. All examples are written in pseudo code.

Decoding to Memory

The decoded image will end up in the output buffer with no gaps between lines.

```
READ_ADDR = start of JPG image
READ_COUNT = size of JPG image;
WRITE_STARTADDR = begin output buffer
WRITE_ENDADDR = end of output buffer
WRITE_OFFSET = 0
WRITE_WIDTH = 0
START_X = 0
START_Y = 0
END_X = 0
END_Y = 0

STATUS.START = 1
Loop
{
  if STATUS.JPGRDY then quit, all went OK
  if STATUS.JPGERR then quit, should not happen
  if STATUS.REMPTY then quit, should not happen
  if STATUS.WFULL then quit, should not happen
}
```

Legacy documentation
refer to the Altium Wiki for current information

Decoding to a Screen Buffer in Memory

The decoded image will end up in the screen buffer with each line starting at WRITE_WIDTH intervals. If lines are longer than WRITE_WIDTH the additional pixels will be discarded.

```
READ_ADDR = start of JPG image
READ_COUNT = size of JPG image;
WRITE_STARTADDR = begin screen buffer
WRITE_ENDADDR = end of screen buffer
WRITE_OFFSET = 0
WRITE_WIDTH = width of screen line
START_X = 0
START_Y = 0
END_X = width of screen
END_Y = height of screen

STATUS.START = 1
Loop
{
  if STATUS.JPGRDY then quit, all went OK
  if STATUS.JPGERR then quit, should not happen
  if STATUS.REMPTY then quit, should not happen
  if STATUS.WFULL then quit, should not happen
}
```

Block-Based Reading of JPEG Data from a Small Buffer

A possible use for this implementation is reading the JPEG image from hard disk, where data transfer takes place in blocks of fixed size.

The decoded image will end up in the output buffer (presumed large enough in this example) with no gaps between lines.

```
READ_ADDR = 0
READ_COUNT = 0;
WRITE_STARTADDR = begin output buffer
WRITE_ENDADDR = end of output buffer
WRITE_OFFSET = 0
WRITE_WIDTH = 0
START_X = 0
START_Y = 0
END_X = 0
END_Y = 0

STATUS.START = 1
Loop
{
  if STATUS.JPGRDY then quit, all went OK
  if STATUS.JPGERR then quit, should not happen
  if STATUS.WFULL then quit, should not happen

  if STATUS.REMPTY then
    read next part from JPG data into read buffer
    READ_ADDR = start of read buffer
    READ_COUNT = size of data read
    STATUS.REMPTY = 1
}
```

Legacy documentation
refer to the Altium Wiki for current information

Block-Based Writing of Decoded Image to a Small Buffer in Fixed Sets of Image Lines

A possible use for this implementation is when running image processing over the decoded image before displaying it on a screen. This way only a buffer for 16 screen lines is needed instead of a buffer for a full screen.

JPEG images are always decoded in batches of 8 or 16 lines at a time, depending on the way they are compressed. For this implementation to be usable under all circumstances a buffer of a multiple of 16 lines is required. If this requirement is met this will guarantee ProcessLines(buffer, line_width, number_of_lines) will be called in top-down order once for each set of 16 (or at the last call possibly less) image lines decoded.

```
READ_ADDR = start of JPG image
READ_COUNT = size of JPG image;
WRITE_STARTADDR = 0
WRITE_ENDADDR = 0
WRITE_OFFSET = 0
WRITE_WIDTH = 0
START_X = 0
START_Y = 0
END_X = 0
END_Y = 0

buffer = void
lines_in_buffer = 16 (can be any multiple of 16)

STATUS.START = 1
Loop
{
  if STATUS.JPGERR then quit, should not happen
  if STATUS.REMPTY then quit, should not happen

  if STATUS.WFULL or STATUS.JPGRDY then
    if buffer = void then
       allocate buffer with size (lines_in_buffer * SIZE_X_EVEN) in pixels
              WRITE_STARTADDR = begin of buffer
              WRITE_ENDADDR = end of buffer
              lines_to_decode = SIZE_Y
    else
       if lines_in_buffer < lines_to_decode then
              lines_in_buffer = lines_to_decode

       lines_to_decode = lines_to_decode - lines_in_buffer;

       call ProcessLines(buffer, SIZE_X_EVEN, lines_in_buffer)
       WRITE_OFFSET = WRITE_OFFSET + size of buffer

    if STATUS.JPGRDY then quit, all went OK

    STATUS.WFULL = 1
}
```

Legacy documentation
refer to the Altium Wiki for current information

Block-Based Writing of Decoded Image using a Small Buffer of Arbitrary Size

A possible use for this implementation is storing the resulting image on hard disk, where data transfer takes place in blocks of fixed size.

The image will decode into a small buffer, which will be swapped with offline storage while decoding. As the buffer size is not a precise multiple of 16 image lines, the decoder will want to write more than once in every buffer sized part of the image. This means that data must be updated two-way with the offline storage, writing only is not sufficient.

```
-- In this code we divide the memory for the decoded image in imaginary
-- blocks the size of the work buffer.
--
-- As the decoder wants to write a pixel, the work buffer is filled with the
-- corresponding block from the full image. As soon as the decoder wants to
-- write a pixel outside the block currently in the work buffer, it halts.
-- We then copy the work buffer back to its place in the full image.
-- This process repeats until the complete image has been decoded.

READ_ADDR = start of JPG image
READ_COUNT = size of JPG image;
WRITE_STARTADDR = start of work buffer
WRITE_ENDADDR = end of work buffer
WRITE_OFFSET = 0
WRITE_WIDTH = 0
START_X = 0
START_Y = 0
END_X = 0
END_Y = 0

mirror offline storage at WRITE_OFFSET to the work buffer

STATUS.START = 1
Loop
{
  if STATUS.JPGERR then quit, should not happen
  if STATUS.REMPTY then quit, should not happen

  if STATUS.WFULL or STATUS.JPGRDY then
    mirror work buffer to offline storage at WRITE_OFFSET

    if STATUS.JPGRDY then quit, all went OK

    -- find location in the image the decoder would want to write:

    image_offset = WRITE_OFFSET + WRITE_ADDR - WRITE_STARTADDR

    -- if we divided the image into multiple areas of the size of
    -- our work buffer, find out which of them the decoder would
    -- want to write:

    offset_blocknumber = floor(image_offset / size_of_workbuffer)

    -- now we know where the block we want to mirror is located:

    WRITE_OFFSET = offset_blocknumber x size_of_workbuffer

    mirror offline storage at WRITE_OFFSET to the work buffer

    STATUS.WFULL = 1
}
```

Legacy documentation
refer to the Altium Wiki for current information

Revision History

Date	Version No.	Revision
18-Oct-2007	1.0	Initial release
07-Mar-2008	2.0	Updated for Altium Designer Summer 08

Software, hardware, documentation and related materials:

Copyright © 2008 Altium Limited.

All rights reserved. You are permitted to print this document provided that (1) the use of such is for personal use only and will not be copied or posted on any network computer or broadcast in any media, and (2) no modifications of the document is made. Unauthorized duplication, in whole or part, of this document by any means, mechanical or electronic, including translation into another language, except for brief excerpts in published reviews, is prohibited without the express written permission of Altium Limited. Unauthorized duplication of this work may also be prohibited by local statute. Violators may be subject to both criminal and civil penalties, including fines and/or imprisonment. Altium, Altium Designer, Board Insight, Design Explorer, DXP, LiveDesign, NanoBoard, NanoTalk, P-CAD, SimCode, Situs, TASKING, and Topological Autorouting and their respective logos are trademarks or registered trademarks of Altium Limited or its subsidiaries. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.