Order Number
JT1001SR/D

PRELIMINARY

JATO TECHNOLOGIES

JT1001 Software Reference Manual

Control Registers

Rev. 0
August 24, 1998

JT1001

Jato Technologies, Inc.
JT1001 Network Controller
Software Reference Manual

TRADEMARKS

JT1001, Propulsion, and CLIP are trademarks of Jato Technologies, Inc.

Magic Packet is a trademark of Advanced Micro Devices, Inc.

Any other trademarks or registered trademarks are the property of their respective owners.

DISCLAIMER

Information in this document is provided in connection with Jato Technologies, Inc. products. No license, express or implied to any intellectual property rights, is granted by this document. While every attempt has been made to assure that the information presented in this document is accurate, Jato Technologies, Inc. assumes no liability whatsoever relating to any possible inaccuracies. Jato Technologies, Inc. assumes no liability whatsoever relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright, or other intellectual property right.

Jato Technologies, Inc. reserves the right to make changes to specifications and product descriptions at any time, without notice.

HOW TO REACH US

Jato Technologies, Inc.
505 E. Huntland Drive, Suite 550
Austin, TX 78752

Telephone (512) 407-2100

Fax (512) 452-5592

http://www.jatotech.com
info@jatotech.com

Table of Contents

List of Figures

List of Tables

Section 1
Overview

The JT1001 controller is designed to provide an optimal combination of cost and system-level performance, both at the file server and at the workstation. The device achieves this optimal combination by exploiting attributes of the PCI bus, by combining large amounts of RAM on-board the device, by leveraging behavioral characteristics of PC system software, and by incorporating a flexible system interface that adapts readily to the shifts in the performance balance between the microprocessor and the other system components. The details of this design are described in this manual and include algorithm and interface descriptions. However, the large scale features of the device can be summarized as follows:

- Deep on-board receive (RX) FIFO (64K).
- Deep on-board transmit (TX) FIFO (32K).
- 64-bit address and data paths.
- Supports three data transfer methods:
  - — Programmed I/O (PIO).
  - — Packet descriptor list (PDL).
  - — Packet Propulsion™ (PDC) I/O method.
- 10/100/1000 Mbps operation.
- Full-duplex and half-duplex operation.
- Supports auto-negotiation of duplex mode and link speed.
- Supports MII, G/MII, and PCS PHY connections.
- PCI power management support.
- Magic Packet™ data sequence wake up.
- VLAN support.
- TCP/IP checksum generation/validation.
- Expansion ROM for remote IPL and BIOS extensions.
- Selectable receive/transmit prioritization.
- Support for big/little endian byte ordering.
- Promiscuous receive mode.
- Independent enable/disable of transmitter and receiver.
- Loopback at MAC and PHY.
- High-level register interface to EEPROM (read/write).
- Flexible PCI register interface to G/MII registers.

- Receive and transmit completion interrupts can be selected on a per-packet basis.
- Programmable high/low watermark interrupts for receive and transmit data FIFOs.
- Error counters for dropped packets, errored packets, late collisions, etc.
- Support for up to four programmable LEDs.

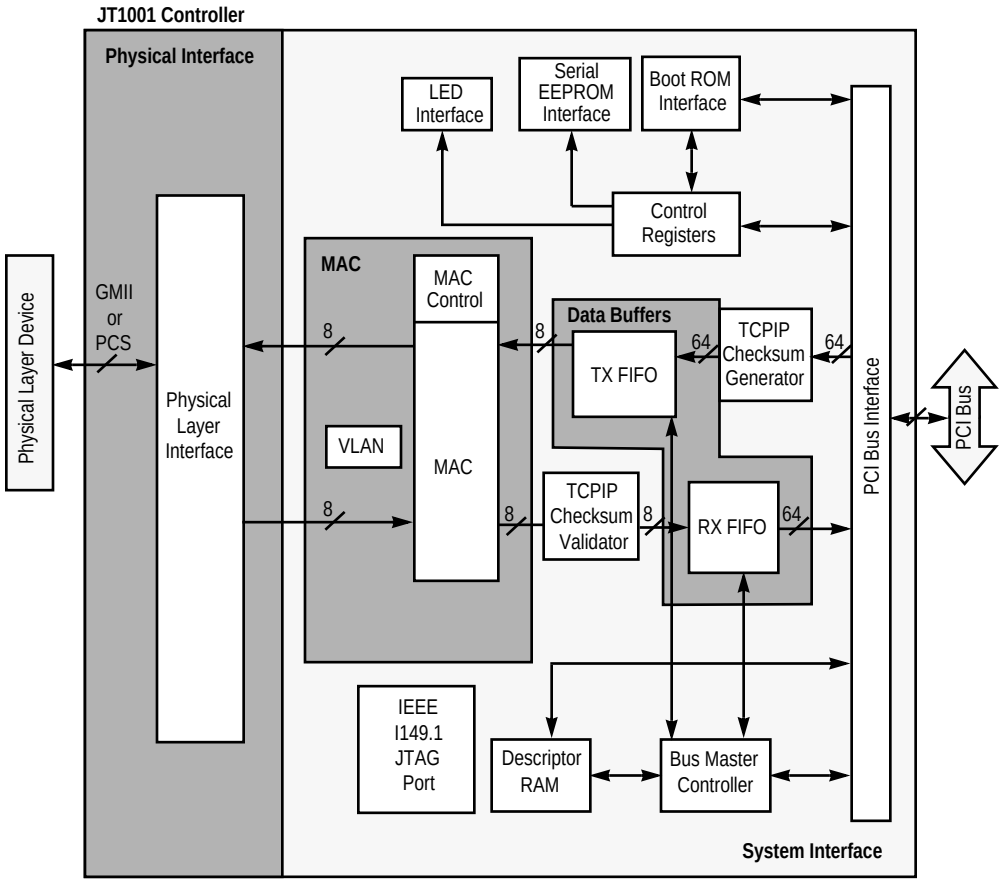


Figure 1-1. JT1001 Block Diagram

Section 2
Types of Registers

Several types of registers are used in the JT1001. They are:

- Control and Status Registers.
  - — Control and Status Registers (CSRs) are accessible by HOST software. CSRs are used to control the operational behavior of the JT1001 controller and ascertain its status. In particular, CSRs can be read/write, read only, write only, or a combination of all three. The read/write attribute of a particular bit or sequence of bits in a CSR is individually set. That is to say, a CSR can be entirely dedicated to one function *(*as is the case with the *Transmit PDL Address Register),* or can be subdivided into one or more bit fields *(*like *Mode Register – 1),* each having its own read/write behavior.
  - — Frequently accessed bit fields in CSRs are implemented as set/reset registers. This type of register is subdivided into one or more mask bits and one or more set/reset control bits. The mask bits determine whether a particular bit will be affected by a given write operation. For example, a mask value of 1001b will allow a write to the least significant bit (LSB) and most significant bit (MSB) of a 4-bit field while preventing writes to the middle bits. If the control bit is set to 1, then the MSB and LSB will have 1s written to them. If the control bit is set to 0, then the MSB and LSB will have 0s written. Again, in both cases the contents of the middle bits remain unchanged. When set/reset registers are read, the set/reset bits are ignored (actually, they are read as 0s) and those bit fields that have an R/W attribute will return their current value. *Mode Register – 1* is an example of a set/reset CSR.
  - — Some CSR bit fields are self-clearing. When set by HOST software, self-clearing bit fields remain set until some activity completes, at which point the bit field is automatically reset by the JT1001. Self-clearing bits can be polled by HOST software to determine when the activity has completed. The SWRE bit in *Mode Register – 1* is an example of an auto-reset bit (field).
  - — Some CSR registers are automatically cleared when read. Typically, this type of behavior is used in counter registers. Once the count is retrieved, the register resets and counting begins again at some predetermined value, usually 0. The *Event Status Register* is an example of a clear when read register.
  - — CSRs are either 32 or 64 bits wide and can be accessed using either I/O or memory cycles.

  - — The 64-bit registers can be accessed 32 bits at a time. However, 64-bit registers that effect an action when read or written must be accessed most significant DWORD (MSD) first. For example, to pass the address of a transmit PDL to the JT1001 controller, the MSD is written first, followed by the least significant DWORD (LSD). When the JT1001 controller detects the write to the LSD of the *Transmit PDL Address Register,* it will assume that all 64 bits have been written and that the operation can begin.
  - — In cases where extensive bit manipulation of CSRs is expected, the most heavily used bits are kept in the low order 16 bits of the register. This is done to accommodate the byte and word oriented operations of x86 microprocessors.
  - — The CSRs are defined in Section 5.
- PCI Configuration Space Registers.
  - — The JT1001 configuration is achieved by partially using PCI configuration space registers and partially using CSRs. Configuration aspects pertaining to system resources such as interrupts and address space are handled in the standard manner using PCI configuration registers. Configuration of the operational characteristics of the JT1001; e.g., wire speed, reception of multicast frames, etc., is done by programming specific values into CSRs.
  - — The PCI configuration registers adhere to the PCI v2.1 Specification and the PCI Power Management Specification. See Section 5 for a map of the configuration registers.
- All of the PCI configuration registers and the CSR registers can be read or written from the PCI bus, EEPROM, or internal blocks.
- As previously noted, the device registers that are visible on the PCI bus can either be I/O mapped or memory mapped in PCI address space.
  - — The registers are all aligned on DWORD boundaries.
  - — The registers support 8-, 16-, 32-, and 64-bit accesses. Note that at this time, the PCI bus only supports 32-bit I/O. All 64-bit accesses are for memory cycles only. Although 64-bit I/O is not currently supported, it is expected that it will be defined in the standard and supported by CPUs within the lifetime of the JT1001.
- Data written to reserved bit fields will be ignored. A potential consequence of this approach is that certain types of HOST software defects may go undetected until the reserved bits are utilized in future revisions of the device.
- The JT1001 CSRs are organized to accommodate high-performance drivers. The registers have been organized to minimize the bit manipulations required for mainstream packet processing.

Section 3
Theory of Operations

3.1 Input/Output Methods

Three I/O methods are defined for the JT1001: programmed input/output (PIO), packet descriptor list (PDL), and packet Propulsion (PDC) I/O method. Of these three methods, PIO and PDL are extensively used in conventional ethernet adapters and are described briefly below. The third technique, packet Propulsion I/O method, is Jato Technologies' unique and proprietary data transfer method designed to highly optimize packet processing for increased I/O bus utilization and data throughput.

3.1.1 Programmed Input/Output

The PIO method implemented in the JT1001 is a traditional I/O method where the CPU moves data into and out of the device. CPU read and write operations (IN and OUT instructions in x86 parlance) are performed to device registers to either place data to be transmitted into the transmit data buffer (TX FIFO) or to extract data from the receive data buffer (RX FIFO). Figure 3-1 depicts the process as it is applied when transmitting a packet. Step (1) is to ascertain whether the packet to be transmitted fits into the TX FIFO (in the diagram, it is assumed that the packet fits). Step (2) is to transfer the data to the device. The CPU performs this task by repeatedly writing DWORDs of packet data to the *TX FIFO Write Register* until all packet data is exhausted. Step (3) is to inform the device that a complete packet has been placed into the TX FIFO. Upon receiving this indication, the device will initiate transmission of the packet at the next available opportunity.

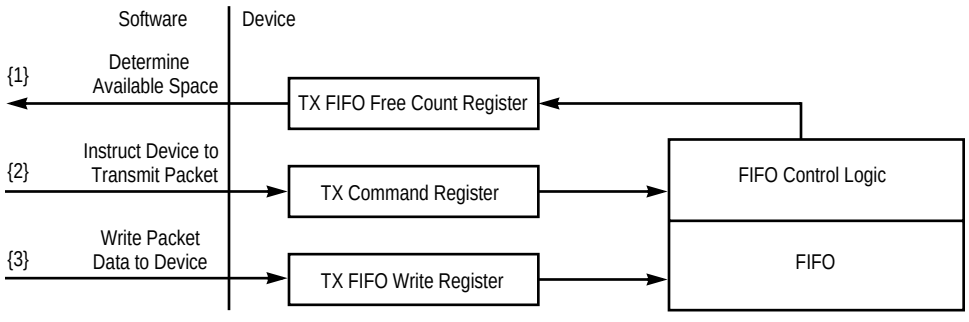


Figure 3-1. PIO Data Transfer Process

3.1.2 Packet Descriptor List

The PDL technique is also commonly referred to as scatter-gather bus master. The PDL is a data structure that is comprised of a header followed by a number of packet fragment descriptors. The header specifies the total length of the packet (the sum of the lengths of the fragments), the number of fragments, and option flags indicating any special processing requirements for the packet. The fragment descriptors specify the physical memory addresses of the buffer fragments and their individual lengths. The data structure is arranged as follows (bit 0 is the LSB):

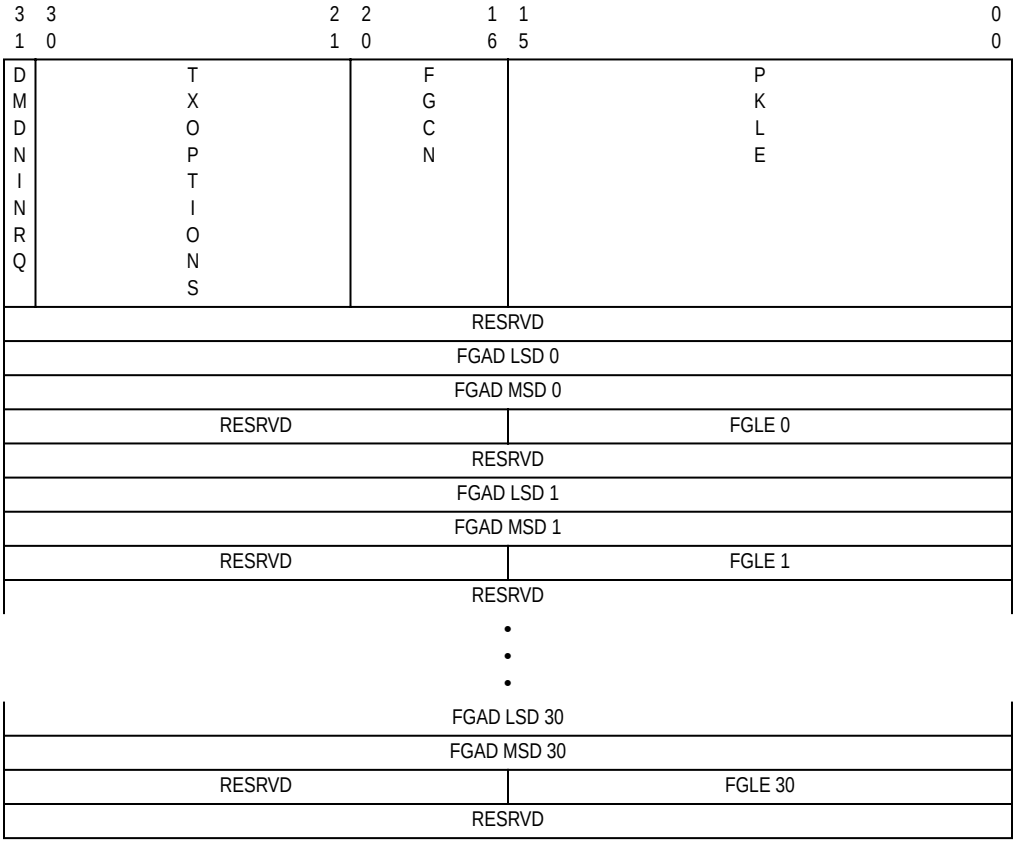


Figure 3-2. Transmit Packet Descriptor List

For clarity, the field names are briefly defined below:

PKLE — *Packet Length.* The sum of the lengths of the individual fragments.

FGCN — *Fragment Count.* The count of fragments defined within the PDL.

TXOPTIONS — *Per Packet Transmission Options.*

DMDNINRQ — *Request for Interrupt Upon Completion of the Packet Transfer to the JT1001 controller.*

FGAD LSD n — *Fragment n Address, Least Significant DWORD.*

FGAD MSD n — *Fragment n Address, Most Significant DWORD.*

FGLE n — *Fragment n Length.*

Details pertaining to this data structure are presented in the section describing the *Transmit PDL Address Register's* LSD. The PDL for receive is virtually identical. It is described in detail in the *Receive PDL Address Register* MSD/LSD sections.

Figure 3-3 depicts the interaction between the JT1001 and its supporting system software when performing PDL DMA transfers. Although the diagram is presented from the vantage point of packet transmission, packet reception behaves in almost the same fashion. The differences will be highlighted at the end of the sub-section.

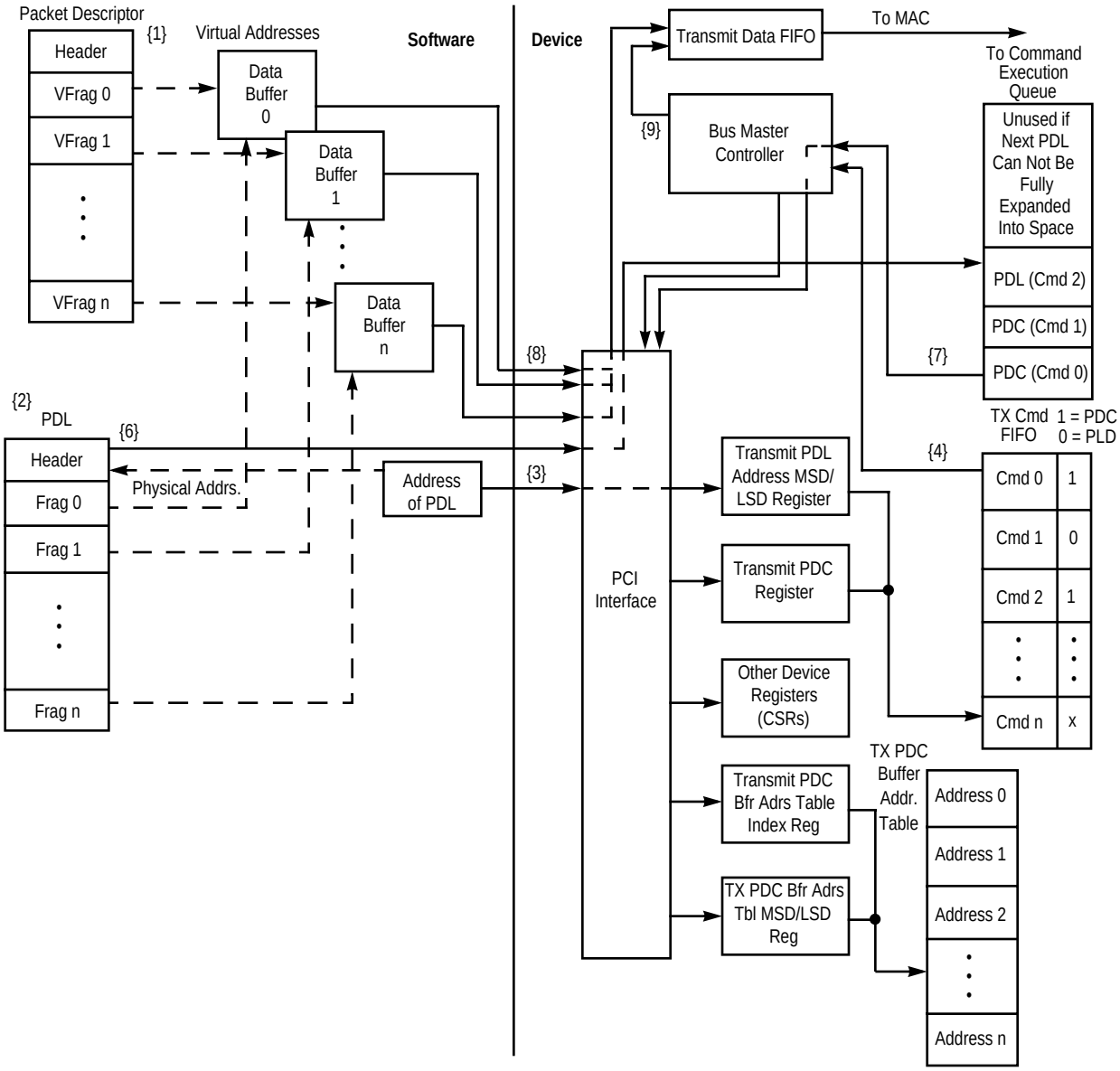


Figure 3-3. PDL Data Transfer Process

In Figure 3-3, the sequence for packet transmission is as follows:

1. An indication is received by the JT1001 system software that a packet is to be transmitted. The indication is accompanied by some form of packet descriptor data structure usually containing multiple packet buffer fragments that are to be sent in the sequence that they are found in the descriptor.

2. The addresses of the packet fragments found in the packet descriptor can be virtual or physical addresses. It is typical for them to be the virtual addresses of buffers constructed by a protocol stack. In this case, virtual addresses must be converted to physical addresses. This operation usually involves a call into the HOST operating system. Once the physical addresses for the packet are known, they are stored in the fragment address fields of the PDL's fragment descriptors.

3. After completely formatting the PDL; i.e., once it fully describes the packet to be transmitted, it can be passed to the JT1001 controller for processing. To do this, the starting physical address of the PDL in HOST memory is written to the *Transmit PDL Address MSD Register/LSD Register*. This action has the effect of placing the PDL's address into the transmit command FIFO.

4. When one or more commands are present in the transmit command FIFO, the bus master JT1001 controller (BMC) is prompted to examine the FIFO and extract the first available command. In our example, the queue is assumed to be empty prior to the transmit request and thus the PDL is acted upon at the first available opportunity.

5. When the BMC looks at the command FIFO, it determines that the command is a PDL. Consequently, the BMC issues a request to the PCI block for the transfer of the PDL data structure from HOST memory to the Command Execution Queue (i.e., the header field and the fragment fields).

6. The PCI block responds to the request by effecting the necessary cycles on the bus to transfer the PDL into the Command Execution Queue.

7. Once the PDL is at the front of the Command Execution Queue, the BMC begins to "execute" it. It does so by interpreting the header, setting up its counters, pointers, etc., and then issuing commands to the PCI block to effect data transfers from HOST memory to the transmit data FIFO. For every fragment in the PDL, the BMC issues one data transfer command to the PCI block.

8. The PCI block transfers each packet fragment, one at a time to the transmit data FIFO.

9. Once the packet is completely transferred to the data FIFO, the BMC signals the data FIFO that a complete packet has been transferred. The FIFO control logic then updates its pointers and signals the MAC that a packet is ready for transmission.

The process for receiving a frame differs from the process described above in two respects:

1. The direction of the data flow is reversed.
2. The receive process is driven by the availability of incoming data and the availability of PDLs in the receive command FIFO. In other words, if a packet arrives and there is no PDL or PDC in the receive command FIFO, then the packet will sit in the receive data FIFO until a PDL or PDC is placed into the receive command FIFO.

3.1.3 Packet Propulsion Method (Packet Descriptor Command)

The PDC method for moving data is a specialization of the traditional PDL technique. As previously noted, the principle notion of PDL is that a bus master device is instructed to obtain a command block from HOST system memory. At a minimum, the command block contains a list of the physical addresses of the packet fragment buffers in HOST memory that are to be copied to the device, the count of packet buffer fragments, and the overall length of the data contained in the fragments (the sum of the lengths of the individual fragments).

The device parses the command block, extracting the address of each block of memory (fragment) to process, and effects a transfer of the said fragment from HOST memory to the device. The device repeats this process for each fragment listed in the PDL until all of the data described by the command is copied to the JT1001 controller for transmission (the direction of the data flow is reversed for receive).

Contemplating the nature of the most important modern operating systems, several key points become apparent:

- They support and practically require virtual memory.
- Devices that initiate data movement transactions across peripheral interconnect busses can not use virtual memory addresses to effect the transfers.
- In terms of performance, the conversion of virtual addresses to physical addresses is an expensive one.

These points are significant primarily because of the sequence of events they impose on bus master devices. Again considering the traditional PDL technique, when data is passed to a bus master device, the corresponding device driver must first perform a virtual to physical address conversion for each of the buffer fragments in the data transfer operation. Moreover, a typical buffer passed to the device is broken up into several buffer fragments. That is, the data to be transferred to the device is segmented into several pieces (typically three or four pieces). These facts result in a situation whereby the cost of converting a virtual address to a physical one can be repeated several times for each block of data that is to be transferred to the device. Given that the conversions are expensive, it is desirable to avoid them.

One method for avoiding the virtual-to-physical address translations is for the device driver to allocate blocks of locked memory during device initialization. The address conversion for these blocks can be performed once — at the point where the memory is allocated — and the physical address can be stored away; e.g., in a queue. Each time that a request to transfer data to the device is received from the upper layers, the device driver could very quickly remove the next available memory block from the queue, copy the data provided by the upper layer into the memory block, format a PDL, and then pass the PDL to the JT1001 controller. This method has the following advantages:

- Virtual to physical address translation is avoided.
- The formatting of the PDL is simplified.
- The amount of data to be transferred to the JT1001 controller in the PDL itself is reduced.

This method for processing DMAs is frequently referred to as “double copy” or “double buffer” DMA. Several observations can be made regarding this technique:

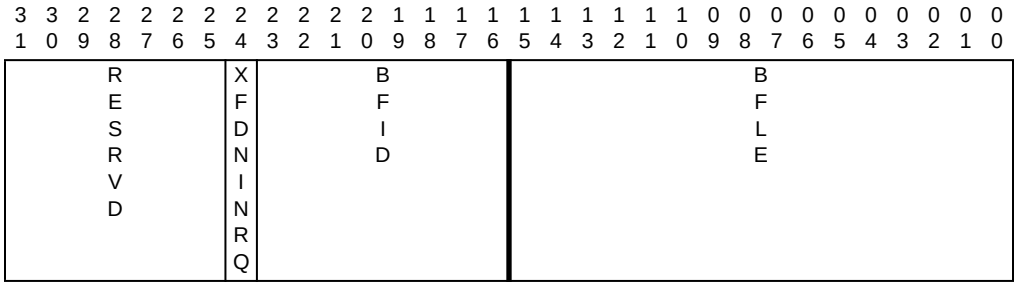
- The fragment count is always 1.
- The length value in the PDL header is the same as the length value of the first fragment.
- The physical address placed into the first fragment address field in the PDL is one of *n* possible physical addresses of pre-allocated locked buffers. Usually, *n* is 16, 32, 64, or some other suitably small integer (i.e., typically *n* <= 128, although it appears that future drivers may begin using values for *n* that are more in the range of 128 <= *n* <= 1024).

Once it is clear that some of the fields in the PDL will always have either the same value, or one value out of a small set of values, an expedited form of double buffering becomes possible. This expedited double buffer technique is called Propulsion technology, or PDC.

The principal ideas behind Propulsion technology are:

- No command block (i.e., PDL) is formatted in HOST memory. Data transfer commands are communicated to the JT1001 controller by passing a packet descriptor command. A PDC is a 32-bit value, subdivided into fields, that completely describes the data transfer operation. A PDC fits entirely within a device register and can be constructed entirely within a CPU register.
- Only one fragment (data buffer) per data transfer operation is communicated to the JT1001 controller using a PDC; i.e., one buffer completely contains all of the data to be transferred to/from the JT1001 controller.
- The address of the data buffer is passed to the JT1001 controller using a small (8-bit) ordinal value that indexes a table maintained on the JT1001 controller. The table has the complete set of physical addresses of buffers allocated by the DRIVER for data transfer purposes.
- The length of the buffer to be copied to/from JT1001 controller is contained within the PDC command.

The actual format of a PDC command is as follows:



Upon closer inspection, one other optimization can be realized with the PDC technique. There is no restriction inherent to the PDC technique that prevents multiple packets from being copied by the HOST (or the JT1001 controller, depending on direction) into the pre-allocated data transfer buffers. In fact, if the data format used for the pre-allocated buffers parallels the data formats used internally by the JT1001 controller, then multiple packets can be easily formatted into the PDC buffers and subsequently transferred to the JT1001 controller with one I/O operation by the HOST (PDC command transfer to the JT1001 controller) and one burst transfer of data by the JT1001 controller.

Thus, Propulsion technique is efficient in its use of bus bandwidth by minimizing the per-buffer overhead associated with the transfer of data to/from the JT1001 controller. Note that the per-buffer overhead includes the number of interactions between HOST software and the JT1001 controller; e.g., interrupts (especially on receive since multiple packets can be delivered into a single PDC and only one interrupt is generated to signal their arrival), command block exchanges, and PIO operations to the device.

Although the PDC technique is very efficient in its use of bus bandwidth, this efficiency does not come without a price. The single most significant drawback of the PDC method is that it requires that the processor move data from application buffers into data transfer buffers — in other words, increased CPU utilization. At first glance, this double copy would seem an insurmountable obstacle to the emergence of PDC as the preferred data transfer technique. However, when one considers that on average many tens, if not hundreds, of CPU clocks are expended in performing virtual-to-physical address translations, and that often times, many such translations are performed per buffer transferred to the JT1001 controller, it becomes evident that a large amount of data can be moved by the CPU in the same amount of time taken for an address translation. Certainly for small data transfers (and virtually half of all data transfers are small), the technique is useful since many tens, if not hundreds, of bytes can be moved in the time it takes to make just one virtual-to-physical address translation. Moreover, as CPUs move to 64-bit and larger word sizes, the efficacy of this technique increases.

Another significant point is that the concern over CPU utilization is not paramount in all systems. Especially in systems where large amounts of data are moved about; e.g., bus utilization is high, and the CPU has nothing else to do, then favoring bus utilization at the expense of CPU utilization can be a desirable trade-off to make. This argument can be extended to multiprocessor

machines where CPU bandwidth outpaces bus bandwidth by a large margin. Here, too, expending CPU utilization to gain bus utilization may be a worthwhile trade-off.

Figure 3-4 depicts the process of data transfer using PDCs.

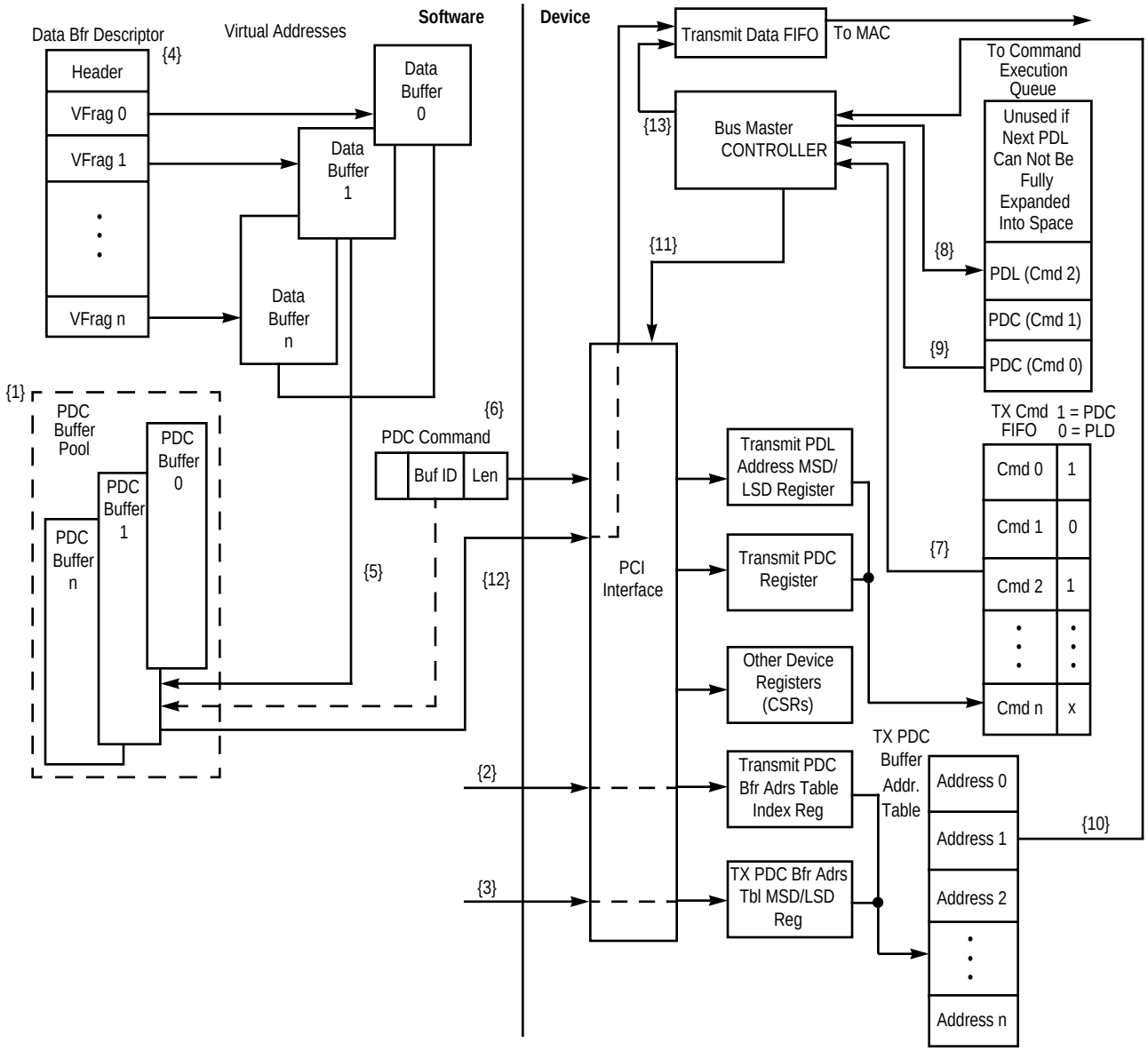


Figure 3-4. PDC Data Transfer Process

The sequence for packet transmission when using PDCs is actually broken into two phases. The first phase happens once during device/driver initialization. It consists of steps 1, 2, and 3, as follows:

1. A pool of locked buffers is allocated from the system. Each buffer is made large enough to accommodate one or more full-size packets. If the individual buffers are made larger than the HOST system's page size (for virtual memory systems), the buffers must be contiguous (i.e., in adjacent pages with the lowest physical address residing in the lowest numbered page).

2. The *Transmit PDC Buffer Address Table Index Register* is pointed to the number 0 Buffer Address Table slot.

3. The addresses of the buffers in the pool are written to the table. Several important points can be mentioned here:

   a. The Transmit PDC Buffer Address Table does not need to be fully utilized. For example, if only two transmit PDC buffers are desired, then only two Transmit PDC Buffer Address Table entries need be used.

   b. Addresses in the table do not need to be in adjacent slots.

   c. Addresses in the table do not need to be in any particular order.

   d. The *Transmit PDC Buffer Address Table Index Register* auto-increments with each write to the *Transmit PDC Buffer Address LSD Register*. This facilitates the writing of strings of buffer addresses to the device. However, the index register can be written prior to every write to the address MSD/LSD registers, thus allowing random write access to the table (both the address and index registers are write only).

The second phase is comprised of steps 4 through 13. These steps happen each time one or more packets are transmitted using a PDC buffer:

4. An indication is received by the JT1001 system software that one or more packets are to be transmitted. The indication is accompanied by some form of packet descriptor data structure usually containing multiple packet buffer fragments (possibly describing multiple packets) that are to be sent in the sequence that they are found in the descriptor.

5. The addresses of the packet fragments found in the packet descriptor can be virtual or physical addresses. It is typical for them to be the virtual addresses of packet buffer fragments constructed by a protocol stack. The driver software responds by allocating (dequeuing) a PDC buffer from the PDC buffer pool and block copying the packet(s) into successive locations within the PDC buffer. Note that in the case where multiple packets are transferred using a single PDC buffer, "fence posts" are inserted between the packets. Details pertaining to the fence posts are provided in the section describing the *Transmit PDC LSD Register*.

6. After copying all of the packet data to a PDC buffer, a PDC is formatted and passed to the JT1001. To do this, the length of the PDC buffer, the buffer ID corresponding to the buffer, and any processing options are formatted into a 32-bit CPU register. The contents of the CPU register are then written to the device's *Transmit PDC Register*. This action has the effect of placing the PDC into the transmit command FIFO and setting a control bit in the FIFO identifying the command as a PDC.

7. When one or more commands are present in the transmit command FIFO, the bus master JT1001 controller (BMC) is prompted to examine the FIFO and extract the first available command. In our example, the queue is assumed to be empty prior to the transmit request and thus the PDC is acted upon at the first available opportunity.

8. When the BMC looks at the command FIFO, it determines that the command is a PDC and moves it directly to the Command Execution Queue.

9. The BMC now takes the PDC command and begins to execute it. It does so by decoding the BFID and BFLE fields (see the *Transmit PDC Register* for a discussion of the PDC command format).

10. The BMC uses the BFID value in the PDC command to index the Transmit PDC Buffer Address Table and obtain the starting physical address of the buffer to be transferred to the JT1001 controller.

11. Once the PDC has been decoded and the starting physical address obtained, the BMC instructs the PCI block to initiate a buffer data transfer to the TX FIFO.

12. The PCI block transfers the buffer data to the TX FIFO at the next available opportunity.

13. Once the packet is completely transferred to the data FIFO, the BMC signals the data FIFO that a complete packet has been transferred. The FIFO control logic then updates its pointers and signals the MAC that a packet is ready for transmission.

3.2 INITIALIZATION

This section contains a discussion of topics related to the initialization of the JT1001. Example pseudo-code is also provided to demonstrate algorithms for initializing the JT1001 controller for PIO, PDL, and PDC I/O methods.

3.2.1 Reset

The JT1001 controller accepts two types of resets: hard reset and soft reset. A hard reset occurs when the PCI $\overline{RST}$ signal is asserted. The JT1001 controller takes the following actions when performing a hard reset:

- All internal state machines of the JT1001 controller are reset to their initial state.
- All internal registers of the JT1001 controller are reset to their default value.

- All CSRs are reset to their default value.
- All PCI configuration space registers are reset to their default value.
- If an EEPROM is present, the JT1001 controller reads the EEPROM and reloads selected CSRs and PCI configuration registers with the values read from EEPROM. For a detailed list of the registers loaded from EEPROM, see Figure 7-1.

A soft reset occurs when HOST software sets the SWRE bit in *Mode Register – 1.* The JT1001 controller takes the same actions for a soft reset as it does for a hard reset with one exception. During a soft reset, the JT1001 controller does not reset the PCI configuration space registers. This is necessary to preserve the hardware resources assigned to the device by system BIOS and/or the operating system.

If HOST software attempts to access the JT1001 controller while a hard or soft reset is in progress, the JT1001 controller generates a PCI retry until the reset has completed. The JT1001 controller indicates a PCI retry to the HOST/PCI bridge by asserting the $\overline{\text{STOP}}$ and deasserting $\overline{\text{TRDY}}$, while keeping $\overline{\text{DEVSEL}}$ asserted during the first data phase of the access. Upon receiving this indication, the HOST/PCI bridge terminates the transaction. After waiting at least two PCI bus cycles, the HOST/PCI bridge will retry the access. The HOST/PCI bridge will continue retrying until the access succeeds. From the perspective of HOST software, the I/O instruction that generated the access blocks until the JT1001 controller's reset completes. To avoid the PCI bus and processor inefficiencies associated with PCI retries, after initiating a soft reset, HOST software should wait 20 ms before it attempts any I/O to the JT1001 controller. Delaying 20 ms allows the JT1001 controller to fully reset without having to issue PCI retries.

3.2.2 Physical Layer Configuration and Status

The JT1001 controller supports three physical layer interfaces: MII, GMII, and PCS. Regardless of the type of PHY present, HOST software interacts with PHY using the *G/MII PHY Access Register.* HOST software accesses PHY registers through this register. HOST software can assume the MII basic register set is present. The basic register set consists of the *Control Register* (register 0) and the *Status Register* (register 1). GMII compliant PHYs have an extended basic register set that includes the *Extended Status Register* (register 15) in addition to the MII basic register set. If a PHY implements the *Extended Status Register,* it sets bit 8 in its *Status Register*. See IEEE Standard 802.3z, clause 22, for a detailed description of the basic, extended basic, and extended register sets. HOST software can also access vendor specific registers on the PHY using the *G/MII PHY Access Register.*

To read the *Status Register* of the PHY at address 2, for example, HOST software writes the following values to the *G/MII PHY Access Register:* GMRRIX = 1, GMCM = 0, GMPHAD = 2. HOST software then polls the register waiting for GMST = 0. When GMST = 0, the JT1001 controller has completed the read operation and GMDA contains the value read from PHY. Write operations occur in a similar manner, except that HOST software puts the value to be written to the PHY register in the GMDA field when it initiates the request to the *G/MII PHY Access Register.*

At initialization time, HOST software is responsible for querying PHY to determine the type of PHY (MII or GMII), the current link speed, and the current duplex mode. Once this is determined, HOST software must set the appropriate values in the *G/MII Mode Register.* Setting the *G/MII Mode Register* determines how the JT1001 controller interacts with PHY when transmitting and receiving packets.

The JT1001 controller has the capability to poll PHY's *Status register* and generate an interrupt when PHY's *Status Register* changes. This capability provides an efficient mechanism to detect changes in the physical layer status. When the interrupt occurs, HOST software can query PHY to determine the exact change in PHY status. HOST software enables this capability by setting the GMSTPOEN bit in *Mode Register – 1* and the PHLASTMS bit in the *Interrupt Mask Register.*

3.2.3 PDC Buffer Allocation

When using the PDC I/O method, HOST software must allocate PDC buffers. HOST software allocates PDC buffers such that they have the following attributes:

- A PDC buffer is physically contiguous. A buffer may span one or more page boundaries as long as the pages are physically contiguous.
- A PDC buffer is locked. The operating system will not swap the buffer to disk or move it a new location in physical memory.
- If possible, the starting address of a receive PDC buffer is on a cache line boundary and the buffer length is evenly divisible by the cache line size. This allows the JT1001 controller to use memory write and invalidate commands when transferring data into the buffer.

When HOST software allocates a PDC buffer, it places the physical address of the PDC into either the Transmit PDC Buffer Address Table or the Receive PDC Buffer Address Table. Each of these tables hold a maximum of 64 PDCs. HOST software does not have to use every entry in the table.

3.2.4 PDL Buffer Allocation

When using the PDL I/O method, HOST software must allocate PDL buffers. HOST software allocates PDL buffers such that they have the following memory attributes:

- A PDL buffer is physically contiguous. A buffer may span a page boundary as long as the pages are physically contiguous.
- A PDL buffer is locked. The operating system will not swap the buffer to disk or move it to a new location in physical memory.
- At a minimum, the starting address of a PDL buffer must begin on a QWORD boundary. If possible, the starting address of PDLs should begin on a cache line boundary and the PDL buffer length is a multiple of cache lines. This allows the JT1001 controller to use memory write and invalidate commands when transferring data into the PDL buffer. The use of memory

write and invalidate commands improves system performance by eliminating unnecessary cache line writes to memory.

The fragment buffers pointed to by a PDL have the same memory attributes as a PDL with the following exception: fragments can be allocated on any byte boundary. This is necessary because fragments are typically allocated by upper layer software and are ephemeral in nature.

3.2.5 System Initialization Event Sequence

At system initialization time, the following sequence of events occurs.

1. The $\overline{\text{RST}}$ signal on the PCI bus is asserted, causing the JT1001 controller to perform a hard reset.
2. HOST BIOS reads and writes the JT1001 controller's *PCI Configuration Space Registers* to determine the JT1001 controller's capabilities and resource requirements.
3. HOST BIOS assigns resources to the JT1001 controller by writing to the JT1001 controller's *PCI Configuration Space Registers*.
4. If an expansion ROM is attached to the JT1001 controller, HOST BIOS shadows (copies) the expansion ROM image into system RAM.
5. If the JT1001 controller is the active boot device, HOST BIOS invokes the expansion ROM image to bring the JT1001 controller to a fully operational state.
6. The operating system is loaded (either from a local disk or via the network connection provided by the JT1001 controller) and HOST BIOS gives control to the operating system.
7. The operating system loads and invokes the HOST device driver software for the JT1001 controller.

3.2.6 Initialization Algorithm

The pseudo-code in Table 3-1 demonstrates a typical algorithm HOST device driver software uses to bring the JT1001 controller to a fully operational state. The "@" in the right-hand column indicates lines where HOST software accesses the JT1001 controller.

Table 3-1. Initialization Pseudo-Code

(1)	**Function Initialize (TransmitPacketList)**	
(2)	Locate the device using PCI services provided by BIOS or the operating system.	@
(3)	Query the CONTROLLER's *PCI Configuration Registers* to determine the IO Base Address, Memory Base Address, and Interrupt level.	@
(4)	Select the desired mode settings via *Mode Register – 1* (CSR00) and *Mode Register – 2* (CSR001)	@
(5)	**If** (the user has configured a locally administered address)	

Table 3-1. Initialization Pseudo-Code (Continued)

```
(6)     Program the locally administered address into the LAN Physical          @
            Address Registers (CSR42 and CSR43).
(7)     Endif
(8)     Query the PHY via G/MII PHY Access Register (CSR44) to determine the link status,   @
            duplex mode, and link speed.
(9)     If (the current PHY mode is incompatible with the link speed, duplex mode, or auto-
            negotiation modes settings the user has requested)
(10)        Reprogram the PHY to the user requested settings using CSR44.        @
(11)        If necessary, force the PHY to renegotiate with its link partner to reflect the   @
                new PHY settings.
(12)    Endif
(13)    Set the appropriate link speed and duplex mode values in the G/MII Mode Register   @
            (CSR45).
(14)    Perform the BIST test using the BIST Register in the PCI Configuration space.   @
(15)    Call LoopbackTest(). See below.                                          @
(16)    If (the loopback test failed)
(17)        Return indicating a fatal error occurred.                            @
(18)    Endif
(19)    If (a bus mastering method will be used to transmit packets)
(20)        Read TXCMFECN from the Command Status Register (CSR51) and save the result   @
                in TxCommandsAvailable.
(21)        Note: PDC and PDL modes are not mutually exclusive.
(22)        If (the PDC I/O method will be used to transmit packets)
(23)            Call InitializePDCTransmit()                                     @
(24)        Endif
(25)        If (the PDL I/O method will be used to transmit packets)
(26)            Call InitializePDLTransmit()
(27)        Endif
(28)    Else
(29)        Read TXFIDWCN from the TX FIFO DWORDs Free Register (CSR21).         @
(30)        Set BytesFreeInTxFIFO = TXFIDWCN * the number of bytes in a DWORD.
(31)    Endif
(32)    If (PDC and/or PDL I/O method will be used for receiving)
(33)        Read RXCMFECN from the Command Status Register (CSR51) and save the result   @
                in RxCommandsAvailable.
(34)        If (the PDC I/O method will be used to receive packets)
(35)            Call InitializePDCReceive(RxCommandsAvailable).                  @
(36)        Endif
(37)        If (the PDL I/O method will be used to receive packets)
(38)            Call InitializePDLReceive(RxCommandsAvailable).                  @
(39)        Endif
(40)        Set the interrupt mask RXPDMS in the Interrupt Mask Register (CSR30). This   @
                will cause an interrupt each time the CONTROLLER has filled a PDL or PDC
                with a inbound packet.
(41)    Else using PIO method
(42)        Set the interrupt mask RXMS in the Interrupt Mask Register (CSR30). This will   @
                cause an interrupt when there is at least one packet in the RX FIFO.
(43)    Endif
(44)    Enable the transmitter and receiver by setting the TXEN and RXEN bits in     @
            Mode Register – 1 (CSR00).
```

Table 3-1. Initialization Pseudo-Code (Continued)

```
(45)      Install the interrupt service routine to handle interrupts generated by the CONTROLLER.
(46)      Enable the CONTROLLER's ability to generate an interrupt by setting the                @
              INENMS bit in the Interrupt Mask Register (CSR30).
(47)      Return Success.
(48)  Endfunction
(49)
(50)  Function LoopbackTest()
(51)      Enable MAC level loopback using the LPBKMD field in Mode Register – 2 (CSR01).        @
(52)      Enable the transmitter and receiver by setting the TXEN and RXEN bits in              @
              Mode Register – 1 (CSR00).
(53)      Transmit a packet to self using PIO method.                                          @
(54)      Poll the RXPKAV bit in the Chip Status Register (CSR11) until it is set.             @
(55)      Receive the packet just transmitted using PIO receive.                               @
(56)      If (errors occurred transmitting or receiving the loopback packet)
(57)          Return indicating PIO loopback test failed.
(58)      Endif
(59)      Build and issue a PDL receive command to the CONTROLLER.                             @
(60)      Build and transmit a packet to self using PDL method.                                @
(61)      Wait for RXDMDNCN field in the Command Status Register (CSR51) to                    @
              be non-zero.
(62)      Examine the packet just received.                                                    @
(63)      If (errors occurred transmitting or receiving the loopback packet)
(64)          Return indicating PDL loopback test failed.
(65)      Endif
(66)      Build and issue a PDC receive command to the CONTROLLER.                             @
(67)      Build and transmit a packet to self using PDC method.                                @
(68)      Wait for RXDMDNCN field in the Command Status Register (CSR51) to                    @
              be non-zero.
(69)      Examine the packet just received.                                                    @
(70)      If (errors occurred transmitting or receiving the loopback packet)
(71)          Return indicating PDC loopback test failed.
(72)      Endif
(73)      Disable the transmitter and receiver by clearing the TXEN and RXEN bits              @
              in Mode Register – 1 (CSR00).
(74)      Return success.
(75)  Endfunction
(76)
(77)  Function InitializePDCTransmit()
(78)      Set the initial table index in the Transmit PDC Buffer Address Table Register (CSR02)
              to zero.
(79)      For (the number of transmit PDC buffers to be allocated — up to the maximum of 64)
(80)          Allocate a transmit PDC buffer.
(81)          Write the physical address of the transmit PDC buffer to the Transmit PDC Buffer  @
                  Address Registers (CSR05 and CSR04). Writing to CSR04 causes the
                  TBIX to automatically increment.
(82)          Add the PDC to the TransmitPDCAvailableList.
(83)      Endfor
(84)  Endfunction
(85)
```

Table 3-1. Initialization Pseudo-Code (Continued)

```
(86)  Function InitializePDCReceive(ReceiveCommandsAvailable)
(87)      Set the initial table index in the Receive PDC Buffer Address Table Register (CSR06)
              to zero.
(88)      For (for the number of PDC buffers to be allocated — up to the maximum of 64)
(89)          Allocate a PDC buffer from the operating system.
(90)          Write the physical address of the receive PDC buffer to the Receive PDC Buffer      @
                  Address Registers (CSR09 and CSR08). Writing to CSR08 causes the TBIX
                  to automatically increment.
(91)          If (RxCommandsAvailable)
(92)              Call PDCQueueReceiveCommand(PDC, ReceiveCommandsAvailable)
                      See Table 3-7
(93)          Else
(94)              Put the PDC on ReceivePDCAvailableList.
(95)          Endif
(96)      Endfor
(97)  Endfunction
(98)
(99)  Function InitializePDLTransmit()
(100)     For (the number of transmit PDL buffers to be allocated)
(101)         Allocate a transmit PDL.
(102)         Put the PDL on the TransmitPDLAvailableList.
(103)     Endfor
(104) Endfunction
(105)
(106) Function InitializePDLReceive(RxCommandsAvailable)
(107)     For (for the number of PDL buffers to be allocated)
(108)         Allocate a PDL buffer from the operating system.
(109)         If (RxCommandsAvailable)
(110)             Allocate a ReceivePacketDescriptor from the operating system.
(111)             Call PDLQueueReceiveCommand (PDL, ReceivePacketDescriptor,                   @
                      RxCommandsAvailable).
                      See Table 3-6
(112)         Else
(113)             Put the PDL on ReceivePDLAvailableList.
(114)         Endif
(115)     Endfor
(116) Endfunction
```

The initialization pseudo-code above and the pseudo-code for packet transmission, packet reception, and interrupt processing that follow constitute a pseudo driver of sorts. The intent of the pseudo driver is to demonstrate the basic concepts of programming the JT1001 controller. It does not represent the only way or the necessarily the optimal way to operate the JT1001 controller.

The pseudo driver is based upon the following set of assumptions and design points:

- The driver will transmit packets using the PIO I/O method or a mixture of PDL/PDC I/O methods. It does not mix the PIO I/O method with other I/O methods.

- The driver receives packets using the PIO mode, PDL, or PDC I/O methods. It does not mix I/O methods when receiving.
- The protocol stack provides a list of packets to transmit rather than one packet.
- The driver indicates received packets to the protocol stack one at a time.

The Initialize() function is the top level function. It is responsible for bringing the device into an operational state. In a real driver, this function is called immediately after the driver is loaded. The major tasks it performs are:

- Locating the device.
- Initializing the PHY.
- Performing a MAC loopback test using the PIO I/O method.
- Allocating PDC buffers and initializing the PDC buffer address tables.
- Allocating PDLs.
- Hooking an interrupt and enabling the JT1001 controller's ability to generate an interrupts for transmit and receive events.

The variables used to model the JT1001 controller's transmitter and receiver states will be discussed in later sections.

3.3 TRANSMIT PACKET PROCESSING

This section contains discussions on topics related to transmitting packets with the JT1001. Example pseudo-code is also provided to demonstrate algorithms for transmitting a packet in PIO, PDL, and PDC modes.

3.3.1 Transmit Packet Padding

By default, when transmitting a packet that is smaller than the minimum packet size, the JT1001 controller adds padding bytes to the end of the packet. For Ethernet, the minimum packet size is 60 bytes, excluding the CRC. The pad bytes added by the JT1001 controller are included in the CRC calculation of the packet. The JT1001 controller's ability to pad undersized packets can be disabled by clearing the TXPPEN bit in *Mode Register – 1*. When this feature is disabled, it is the responsibility of HOST software to pad the packets prior to giving them to the JT1001 controller for transmission. Failure to do so results in runt packets being transmitted on the network.

3.3.2 VLAN Tag Header Insertion

The JT1001 controller provides the capability to insert VLAN tag headers during the transmission of packets. See Section 3.9 for a detailed description of how to use this function.

3.3.3 CRC Generation

By default, the JT1001 controller calculates and appends a 4-byte CRC to outbound packets. This capability can be disabled by clearing the TXCREN bit

in *Mode Register – 1*. When TXCREN is cleared, it is the responsibility of HOST software to include a CRC in the packet data given to the JT1001 controller.

HOST software should ensure the TXCREN bit is set whenever it has enabled other JT1001 controller features that cause the JT1001 controller to insert or modify packet data prior to the packet's transmission. In particular, the TXCREN bit should be set when HOST software has enabled VLAN tag header insertion, TCP/IP checksum insertion, or transmit packet padding. Failure to set the TXCREN in these circumstances results in a packet containing an invalid CRC to be transmitted onto the network.

3.3.4 Transmit Completion Status

The JT1001 implements a "lying send" transmit policy. This means a packet is considered to be successfully transmitted as soon as it is copied into the JT1001 controller's TX FIFO. When using the PIO I/O method, this occurs as soon as HOST software has moved the packet into the TX FIFO. When using the PDL and PDC I/O methods, this occurs as soon as the JT1001 controller has transferred the packet data into the TX FIFO. Ultimately, it is the responsibility of the protocols above the driver to ensure that packets are successfully transmitted to remote stations. If a packet is lost during transmission by the JT1001 controller, the protocol is responsible for recognizing that the packet is lost and effecting a corrective action (e.g., retransmit).

3.3.5 Transmit Statistics

The JT1001 controller maintains the following packet transmission statistics: aFramesTransmittedOK, aSingleCollisionFrames, aMultipleCollisionFrames, Errored Transmit Packet Count, TCP/IP Non Ipv4 Packet Count, and Late Collision Count. The counts do not wrap. See Table 5-1 for a detailed description of these statistics.

3.3.6 Simultaneous Use of PDL, PDC, and PIO I/O Methods

The JT1001 controller supports the use of PDL and PDC I/O methods simultaneously. When transferring data, HOST software indicates the desired data transfer method by the CSR used to initiate the transfer. For packet transmission using the PDC I/O method, HOST software initiates the process by writing to the *Transmit PDC Register*. If the HOST wishes to transmit a packet using the PDL method, it writes to the *Transmit PDL Address Register* instead.

Although intermixing of PDC and PDL transmit commands is directly supported by the JT1001 controller, intermixing of PIO with either PDC or PDL transfer methods is **not** directly supported. It is possible to intermix PIO with the other two transfer methods, however, careful coordination must be carried out to prevent simultaneous accesses to the TX FIFO by the JT1001 controller's System Interface Block and HOST software. More specifically, prior to initiating a transmission using the PIO method, HOST software must guarantee that all PDL and/or PDC transmit commands issued have been completed by the JT1001 controller. A PDL and PDC transmit command is considered completed when the JT1001 controller has transferred the transmit packet data from HOST

memory into the JT1001 controller's TX FIFO and incremented the TXDMDNCN count in the *Command Status Register*.

3.3.7 Programmed Input/Output Method of Transmission

PIO mode is often referred to as a slave mode. The two terms are used interchangeably in this document. In PIO mode, the HOST is responsible for effecting all packet data movement to and from the JT1001 controller.

Transmitting a packet using the PIO method is a four-step process:

1. Determining the TX FIFO has enough free space to accommodate the transmit header and packet.
2. Writing the transmit header to the TX FIFO.
3. Writing the packet data to the TX FIFO.
4. Issuing the transmit command.

The JT1001 controller maintains a count of the number of free DWORDs in the TX FIFO. The JT1001 controller decrements the count as data is written to the FIFO, and increments the count as data is removed from the FIFO and transmitted. HOST software ascertains this count by reading the *TX FIFO DWORDs Free Register*. If the TX FIFO does not contain enough free space to accommodate the packet, HOST software must wait until enough free space exists. HOST software waits by polling, retrying periodically, or by requesting an interrupt be generated when the TX FIFO hits a low watermark. Refer to the *TX FIFO Low Watermark Register* description for more information on how to generate a TX FIFO low watermark interrupt.

Once HOST software has determined the TX FIFO can accommodate the packet, it constructs the transmit header and writes it to the TX FIFO via the *TX FIFO Write Register*. Next, HOST software copies the data to the JT1001 controller's TX FIFO by sequencing through the packet data and writing it to the *TX FIFO Write Register*. HOST software then sets the SLMDTXCM bit in the *Command Register* to indicate the entire packet is in the TX FIFO and is ready for transmission. The JT1001 controller transmits the packet data onto the network in the order that it is written to the *TX FIFO Write Register*.

The pseudo-code in Table 3-2 demonstrates how to transmit a list of packets using the PIO data transfer method. Lines with an "@" in the right-hand column indicate an access to the JT1001 controller.

Table 3-2. PIO Transmit Pseudo-Code

```
(1)  Function PIOTransmitPacketList(TransmitPacketList)
(2)      Get the first packet from TransmitPacketList.
(3)      While (there is a packet to be transmitted)
(4)          PacketLength = the number of bytes in the packet.
(5)          Set RetryCount = MAX_RETRIES + 1.
(6)          While (BytesFreeInTxFIFO < PacketLength + number of bytes in the
                 transmit header)
```

Table 3-2. PIO Transmit Pseudo-Code (Continued)

```
(7)         If (RetryCount = 0)
(8)             Set the transmit status code for the current and all remaining packets
                    in TransmitPacketList to indicate they did not transmit.
(9)             Return out of TX FIFO resources.
(10)        Endif
(11)        Read TXFIDWCN from the TX FIFO DWORDs Free Register (CSR21).                @
(12)        Set BytesFreeInTxFIFO = TXFIDWCN * the number of bytes in a DWORD.
(13)        Decrement RetryCount.
(14)    Endwhile
(15)    Determine the per packet processing options.
(16)    Construct the FIFO Transmit Header using the PacketLength and per packet
            processing options.
(17)    Write the FIFO Transmit Header to the TX FIFO Write Register (CSR22).       @
(18)    For (each fragment in the packet)
(19)        If (the fragment starting address is odd)
(20)            Write the first BYTE of the fragment to the TX FIFO Write           @
                    Register (CSR22).
(21)        Endif
(22)        If (the fragment starting address is not evenly divisible by the size of a DWORD)
(23)            Write the next WORD of the fragment to the Transmit FIFO Write Register  @
                    (CSR22).
(24)        Endif
(25)        While (at least a DWORD remains in the fragment)
(26)            Write the next DWORD of the fragment to the Transmit  FIFO Write   @
                    Register (CSR22).
(27)        Endwhile
(28)        If (at least a WORD is remains in the fragment)
(29)            Write the next WORD of the fragment to the Transmit FIFO Write Register  @
                    (CSR22).
(30)            Decrement the remainder by the size of a WORD.
(31)        Endif
(32)        If (a BYTE remains in the fragment)
(33)            Write the next BYTE of the fragment to the TX FIFO  Write Register   @
                    (CSR22).
(34)        Endif
(35)    Endfor
(36)    Decrement BytesFreeInTxFIFO by (the number of bytes in the packet rounded
            up to the next multiple of 8) + the size of the transmit header.
(37)    Start the PIO transmit by setting the SLMDTXCM in the Command Register      @
            (CSR29).
(38)    Set the packet's transmit status code to success.
(39)    Get the next packet in the TransmitPacketList.
(40)  Endwhile
(41)  Return Success.
(42) Endfunction
```

The PIO transmit pseudo-code models the JT1001 controller's transmitter using the variable BytesFreeInTxFIFO. This variable represents the minimum number of unused bytes in the TX FIFO at any given point in time. The variable is used

to determine if there is enough space in the TX FIFO to accommodate a packet and its transmit header. Initially, BytesFreeInTxFIFO is set to the size of the TX FIFO. Each time a packet is copied into the TX FIFO, BytesFreeInTxFIFO decrements by the number of bytes in the packet plus the size of the transmit header rounded up to the next QWORD boundary.

3.3.8 Packet Descriptor List Method of Transmission

The pseudo-code in Table 3-3 demonstrates how to transmit a list of packets using the PDL data transfer method. Lines with an "@" in the right-hand column indicate an access to the JT1001 controller.

Table 3-3. PDL Transmit Pseudo-Code

```
(1)  Function PDLTransmitPacketList(TransmitPacketList)
(2)      Get the first packet from TransmitPacketList.
(3)      While (there is a packet to be transmitted)
(4)          If (TransmitCommandsAvailable = 0)
(5)              Read TXCMFECN from the Command Status Register (CSR51).        @
(6)              If (TXCMFECN = 0)
(7)                  Set the transmit status code for the current and all remaining packets
                         in TransmitPacketList to indicate they did not transmit.
(8)                  Return indicating no more transmit commands are available.
(9)              Else
(10)                 TransmitCommandsAvailable = the value read from TXCMFECN.
(11)             Endif
(12)         Endif
(13)         If (a PDL buffer is available)
(14)             Get a PDL from the TransmitPDLAvailableList.
(15)             TotalLength = 0.
(16)             PDLFragmentIndex = 0.
(17)             For (each fragment in the packet)
(18)                 If (the fragment is not in locked memory)
(19)                     Call the operating system to lock the memory.
(20)                 Endif
(21)                 If (the fragment address is a virtual address)
(22)                     Call operating system to convert the virtual address to a list of
                             physical addresses.
(23)                 Endif
(24)                 For (each of the virtual fragment's physical addresses)
(25)                     Set PDL.FGAD[PDLFragmentIndex] to the physical fragment
                             address.
(26)                     Set PDL.FGLE[PDLFragmentIndex] to the number of bytes in the
                             physical fragment.
(27)                     TotalLength = TotalLength + the number of bytes in the physical
                             fragment.
(28)                     Move to the next physical fragment in the physical address list.
(29)                     Increment PDLFragmentIndex.
(30)                 Endfor
(31)             Endfor
```

Table 3-3. PDL Transmit Pseudo-Code (Continued)

```
(32)            Set PDL.PKLE field to the TotalLength calculated.
(33)            Set PDL.FGCN field to PDLFragmentIndex.
(34)            Set the desired per packet processing options in the PDL header.
(35)            Queue the PDL onto the TransmitCommandsInProgressQueue.
(36)            Write the MSD of the PDL's physical address to Transmit PDL          @
                    Address MSD Register (CSR13).
(37)            Write the LSD of the PDL's physical address to Transmit PDL          @
                    Address LSD Register (CSR12).
(38)            Set the transmit status code status for the packet to indicate the transmit is
                    in progress.
(39)            Decrement TransmitCommandsAvailable.
(40)        Else
(41)            Set the transmit status code for the current and all remaining packets in
                    TransmitPacketList to indicate they did not  ransmit.
(42)            Return indicating no more PDLs are available.
(43)        Endif
(44)        Get the next packet in the TransmitPacketList.
(45)    Endwhile
(46)    Return Success
(47) Endfunction
(48)
(49) Function PDLTransmitDMADoneEvent(PDL)
(50)    Call the protocol stack and indicate the packet associated with this PDL was transmitted
            successfully.
(51)    Queue the PDL on the TransmitPDLAvailableList.
(52) Endfunction
```

The PDL transmit pseudo-code models the JT1001 controller's transmitter state using the following variables:

- TransmitCommandsAvailable — This variable is the minimum number of additional transmit commands the JT1001 controller can take at any given point in time. Each time a transmit PDL or PDC command is given to the JT1001 controller, this count decrements. When it reaches 0, the count is refreshed by reading the TXCMFECN field in the *Command Status Register*.
- TransmitPDLAvailableList — This is a list of PDLs available for transmitting packets. This pool of transmit PDLs is allocated at initialization time. Each time a packet is transmitted using the PDL I/O method, a PDL is removed from this list. PDLs are returned to this list after a PDL transmit command completes.
- TransmitCommandsInProgressQueue — This is a FIFO queue of PDL and PDC commands issued to the JT1001 controller. This queue preserves the ordering in which the commands were issued to the JT1001 controller. When a transmit PDL or PDC command is given to the JT1001 controller, the PDL or PDC is enqueued on this queue. Items are taken off this queue when a TXDMDNIN interrupt occurs.

The PDLTransmitPacketList() function transmits a list of packets it is given as input. For each packet in the list, it performs the following major tasks:

- Determines if the JT1001 controller can accept any more transmit commands.
- Prepares the fragments that constitute a packet to be transmitted for the JT1001 controller. Preparing the fragments involves:
  - Locking the fragments memory.
  - Acquiring the physical addresses of the fragments.
- Constructs a PDL using the physical addresses and lengths of the fragments.
- Writes the physical address of the PDL to the JT1001 controller's *Transmit PDL Address Registers* (CSR13 and CSR12).

PDLTransmitDMADoneEvent() is called by the interrupt handler each time the JT1001 controller has completed processing for a transmit PDL. When called, it notifies the protocol stack that the packet has been transmitted successfully and returns the PDL associated with the packet back to the list of available transmit PDLs.

3.3.9 Packet Propulsion Mode Method of Transmission

The pseudo-code in Table 3-4 demonstrates how to transmit a list of packets using the PDC data transfer method. Lines with an "@" in the right-hand column indicate an access to the JT1001 controller.

Table 3-4. PDC Transmit Pseudo-Code

```
(1)  Function PDCTransmitPacketList(TransmitPacketList)
(2)      Set Offset = size of PDC Buffer to cause a PDC to be obtained during the first iteration of
             the loop.
(3)      Set PDC = NULL to indicate there is not a PDC awaiting ready to be given to the
             CONTROLLER.
(4)      Get the first packet from TransmitPacketList.
(5)      While (there is a packet to be transmitted)
(6)          If (TransmitCommandsAvailable = 0)
(7)              Read TXCMFECN from the Command Status Register (CSR51).                    @
(8)              If (TXCMFECN = 0)
(9)                  Set the transmit status code for the current and all remaining packets in
                         TransmitPacketList the list to indicate they did not transmit.
(10)                 Return indicating no more transmit commands are available.
(11)             Else
(12)                 TransmitCommandsAvailable = the value read from TXCMFECN.
(13)             Endif
(14)         Endif
(15)         PacketLength = the number of bytes in the packet.
(16)         If (the size of the PDC buffer – Offset < PacketLength + the size of the PDC
                 transmit header)
(17)             If (PDC is not NULL)
(18)                 Call StartPDCTransmit(Offset, PDC's Buffer ID).
```

Table 3-4. PDC Transmit Pseudo-Code (Continued)

```
(19)                Endif
(20)                If (a PDC structure is available)
(21)                    Get a PDC from the TransmitPDCAvailableList.
(22)                    HeaderOffset = 0.
(23)                Else
(24)                    Set the transmit status code for the current and all remaining packets in
                            TransmitPacketList the list to indicate they did not transmit.
(25)                    Return indicating no more PDCs are available.
(26)                Endif
(27)            Else
(28)                HeaderOffset = Offset.
(29)            Endif
(30)            Offset = HeaderOffset + the size of the PDC transmit header.
(31)            For (each fragment in the packet)
(32)                Copy fragment data to PDC[Offset].
(33)                Offset = Offset + the number of bytes in the fragment.
(34)            Endfor
(35)            Set the desired packet processing options in the PDC[HeaderOffset].
(36)            Set PDC[HeaderOffset].LEN = PacketLength.
(37)            Offset = HeaderOffset + size of the PDC transmit header + PacketLength.
(38)            Round up the Offset to the next QWORD boundary.
(39)            Set the packet's transmit status code to success.
(40)            Get the next packet in the TransmitPacketList.
(41)        Endwhile
(42)        If (PDC is not NULL)
(43)            Call StartPDCTransmit(Offset, PDC's Buffer ID).
(44)        Endif
(45)        Return Success.
(46)    Endfunction
(47)
(48)    Function StartPDCTransmit(ActualPDCLength, PDCBufferID)
(49)        If (interrupt wanted after data transfer from PDC has completed)
(50)            Set the TransferDoneInterruptFlag.
(51)        Endif
(52)        Construct the PDC Transmit Command with the ActualPDCLength, the
                PDCBufferID, and the TransferDoneInterrupt flag.
(53)        Queue the PDC onto the TransmitCommandsInProgressQueue.
(54)        Write the PDC Transmit Command to the Transmit PDC Register (CSR16).          @
(55)        Decrement TransmitCommandsAvailable.
(56)    Endfunction
(57)
(58)    Function PDCTransmitDMADoneEvent(PDC)
(59)        Queue the PDC on the TransmitPDCAvailableList.
(60)    Endfunction
```

The PDC transmit pseudo-code models the JT1001 controller's transmitter state using the following variables:

- TransmitCommandsAvailable — This variable is the minimum number of additional transmit commands the JT1001 controller can take at any given point in time. Each time a transmit PDL or PDC command is given to the JT1001 controller, this count decrements. When it reaches 0, the count is refreshed by reading the TXCMFECN field in the *Command Status Register.*
- TransmitPDCAvailableList — This is a list of PDCs available for transmitting packets. This pool of transmit PDCs is allocated at initialization time. Each time a packet is transmitted using the PDC I/O method, a PDC is removed from this list. PDCs are returned to this list after a transmit PDC command has completed.
- TransmitCommandsInProgressQueue — This is a FIFO queue of PDL and PDC commands issued to the JT1001 controller. This queue preserves the ordering in which the commands were issued to the JT1001 controller. When a transmit PDL or PDC command is given to the JT1001 controller, it is enqueued on this queue. Items are taken off this queue when a TXDMDNIN interrupt occurs.

PDCTransmitPacketList() transmits a list of packets it is given as input. It performs the following major tasks:

- Determines if the JT1001 controller can accept more transmit commands.
- Gets a PDC from the pool of available PDCs.
- Copies as many packets as it can into the PDC.
- When the PDC can not hold the next packet in the list, the PDC is added to the TransmitCommandsInProgressQueue and is given to the JT1001 controller by writing the PDC's index in the Transmit PDC Address Table to the *Transmit PDC Register* (CSR16).
- The above steps are repeated until all packets are transmitted, no more PDCs are available, or the JT1001 controller can not accept any more commands.

Since PDCTransmitPacketList() copies the packet data into a transmit PDC buffer, it reports the packet as having been transmitted successfully without having to wait for the TXMDNIN to occur.

PDCTransmitDMADoneEvent() is called by the interrupt handler each time the JT1001 controller has completed processing for a transmit PDC. When called, it returns the PDC to the list of available transmit PDCs.

3.4 RECEIVE PACKET PROCESSING

This section contains discussions on topics related to receiving packets with the JT1001. Example pseudo-code is also provided to demonstrate algorithms for transmitting a packet in PIO, PDL, and PDC modes.

3.4.1 Packet Reception Filters

The JT1001 controller provides packet reception filters that can be applied to received packets to determine whether or not an inbound packet is transferred to HOST memory and indicated to HOST software. The packet reception filters operate concurrently with respect to one another and the reception of the packet. Conceptually, however, the filters operate in a hierarchical manner. A packet must pass each active filter in the hierarchy before it is transferred to HOST memory and indicated to HOST software.

The first filter in the hierarchy is the destination address filter. The destination address filter accepts/rejects frames based on the 6-byte destination address in a received packet's MAC header. The acceptance of unicast, broadcast, and multicast packets can be enabled and disabled independently via the UCEN, BCEN, and MCEN bits in *Mode Register – 1*. When UCEN is set, the packet will pass the destination address filter if the destination address matches the address in the *LAN Physical Address Registers*. When the BCEN bit is set, packets containing the all stations broadcast address will pass the destination address filter. When the MCEN bit is set, the JT1001 controller performs a hash operation on the destination address field of multicast packets. The result of the hash operation is an index into a 64-bit hash table. If the hash table bit at the index is set, the packet will pass the destination address filter, otherwise the packet is rejected. The JT1001 controller can be configured to operate in promiscuous mode by setting the POEN bit in *Mode Register – 1*. In promiscuous mode, all packets pass the destination address filter. See the definitions for *Mode Register – 1* and the *Multicast Hash Table LSD/MSD Register* for more details on configuring the destination filter.

The second filter in the hierarchy is the VLAN tag filter. The VLAN tag filter only affects received packets containing a VLAN tag header. When configured by HOST software, the VLAN tag filter passes packets containing a VLAN tag header that matches the protocol ID value set in the *VLAN Tag Protocol ID Register* and a tag control information (TCI) field in the VLAN Tag TCI Table. See Section 3.9 for a detailed description of the JT1001 controller's VLAN support.

The third filter in the hierarchy is the errored packet filter. By default, the errored packet filter is enabled and rejects received packets for which the JT1001 controller detects an error. The JT1001 controller detects CRC, alignment, runt, length, and large packet errors. HOST software can change this behavior by setting the PAERPKEN bit in *Mode Register – 1*. When PAERPKEN is set, the JT1001 controller allows HOST software to receive packets containing errors. In this case, the JT1001 controller will set the appropriate error status bits in the packet's receive header. Regardless of the state of the PAERPKEN bit, the JT1001 controller always forwards received packets that result in an overflow error (EROV).

The final filter is the TCP/IP checksum filter. When enabled by HOST software, the JT1001 controller rejects received packets containing TCP/IP checksum errors. See Section 3.10 for a detailed description of the JT1001 controller's TCP/IP checksum support.

3.4.2 Packet Receive Status

CRC, runt packet, alignment, and long packet errors are detected by the JT1001 and are signaled to HOST software by way of specific bits in the PIO/PDL/PDC Receive header. The packet receive header also contains additional information bits pertaining to the type of destination address the packet contained, whether or not the packet contained valid TCP/IP checksums, and whether or not the packet contained a VLAN tag. For details on the PIO receive header, see Figure 5-8. For details on the PDL receive header, see Figures 5-2 and 5-3. For details on the PDC receive header, see Figure 5-5.

3.4.3 Receive Statistics

The JT1001 controller maintains the following packet reception statistics: aFramesReceivedOK, aFrameCheckSequenceErrors, aAlignmentErrors, Dropped Packet Count, Errored Receive Packet Count, Runt Packet Count, Large Packet Count, VLAN Accepted Packet Count, TCP/IP Checksum Error Count, and VLAN Discarded Packet Count. Refer to Table 5-1 for a detailed description of these statistics.

3.4.4 Large Packet Reception

By default, the JT1001 controller regards a packet that exceeds the maximum packet size as an error. Such frames will not be seen by HOST software unless the PAERPKEN bit is set in *Mode Register – 1*. If the PAERPKEN bit is set, the JT1001 controller passes the packet to HOST software and sets the ERROR and LGPK status bits in the packet's receive header to indicate the packet contains an error.

By setting the LGPKEN bit in *Mode Register – 1*, HOST software can modify the JT1001 controller's large packet processing. When LGPKEN is set, the JT1001 controller does not regard a packet that exceeds the maximum packet size as an error. In this case, the JT1001 controller passes large packets to HOST software, regardless of the state of the PAERPKEN bit, and sets the LGPK bit in the packet's receive header.

3.4.5 Simultaneous Use of PDL, PDC, and PIO I/O Methods

The JT1001 controller supports the use of PDL and PDC I/O methods simultaneously. When transferring data, HOST software indicates the desired data transfer method by the CSR used to initiate the transfer. For packet reception using the PDC I/O method, HOST software initiates the process by writing to the *Receive PDC Register*. If the HOST wishes to receive a packet using the PDL method, it writes to the *Receive PDL Address Register* instead.

Although intermixing of PDC and PDL receive commands is directly supported by the JT1001 controller, intermixing of PIO with either PDC or PDL transfer methods is **not** directly supported. It is possible to intermix PIO with the other two transfer methods, however, careful coordination must be carried out to prevent inadvertent simultaneous accesses by the JT1001 controller's system interface block and HOST software. More specifically, prior to initiating a receive using the PIO method, HOST software must guarantee that all PDL and/or PDC receive commands have been completely processed by the JT1001 controller. A PDL and PDC receive command is considered completely processed when the JT1001 controller has transferred the receive packet data from the JT1001 controller's RX FIFO into HOST memory and incremented the RXDMDNCN count in the *Command Status Register*.

3.5 PROGRAMMED INPUT/OUTPUT (PIO) METHOD OF RECEPTION

The pseudo-code in Table 3-5 demonstrates how to receive a packet using the PIO data transfer method. Lines with an "@" in the right-hand column indicate an access to the JT1001 controller.

Table 3-5. PIO Receive Pseudo-Code

```
(1)  Function PIOProcessReceiveEvent()
(2)      Read the RX FIFO Packet Count Register (CSR28) to see how many packets have      @
             been received and save the result in ReceivePacketsAvailable.
(3)      While (ReceivePacketsAvailable > 0)
(4)          While (ReceivePacketsAvailable > 0)
(5)              Read the RX FIFO Read Register (CSR24) to get the first DWORD of         @
                     the Receive Header.
(6)              PacketLength = LEN field in the first DWORD of the Receive  Header.
(7)              Read the RX FIFO Read Register (CSR24) to get the second DWORD of        @
                     the Receive Header.
(8)              Set NumDWORDS = (PacketLength + size of a DWORD – 1)/ size of a DWORD.
(9)              Acquire a receive buffer large enough to hold NumDWORDS of  receive data.
(10)             If (no receive buffers are available)
(11)                 Set the RXFISKPK in the Command Register (CSR29) to skip the         @
                         partially read packet.
(12)                 Return (indicating out of receive buffer resources).
(13)             Endif
(14)             Set the ReceivePacketBuffer to the address of the receive buffer just  allocated.
(15)             For (NumDWORDS)
(16)                 Read a DWORD from the RX FIFO Read Register (CSR24) into             @
                         the ReceivePacketBuffer.
(17)                 Advance the ReceivePacketBuffer pointer by one  DWORD.
(18)             Endfor
(19)             If (NumDWORDS is an odd number)
(20)                 Read a DWORD from the RX FIFO Read Register (CSR24) and discard      @
                         the value read. This is necessary to keep the FIFO QWORD aligned.
(21)             Endif
(22)             Call the protocol stack and give it the ReceivePacketBuffer.
(23)             If (the protocol stack has copied the data)
```

Table 3-5. PIO Receive Pseudo-Code (Continued)

```
(24)                Free the ReceivePacketBuffer.
(25)            Endif
(26)        Decrement the ReceivePacketsAvailable count.
(27)        Endwhile
(28)    Read the RX FIFO Packet Count Register (CSR28) to see how many packets have been    @
            received and save the result in ReceivePacketsAvailable.
(29)    Endwhile
(30)    Return Success.
(31) Endfunction
```

The PIO receive pseudo-code is very simple. The RXMS bit in the *Interrupt Mask Register* is set at initialization time. Setting this bit causes the JT1001 controller to generate an interrupt each time a complete packet has been put in the RX FIFO. Upon determining that an RXIN interrupt event has occurred, the interrupt handler dispatches PIOProcessReceiveEvent(). PIOProcessReceiveEvent() uses the *RX FIFO Read Register* (CSR24) to read the packet's receive header from the FIFO. The receive header contains the length of the packet. The function then performs an even number of DWORD reads of CSR24 and puts the data read into HOST memory. The function proceeds in this manner until the *RX FIFO Packet Count Register* (CSR28) indicates no more packets are in the RX FIFO.

3.6 Packet Descriptor List Method of Reception

The pseudo-code in Table 3-6 demonstrates how to receive a packet using the PDL data transfer method. Lines with an "@" in the right-hand column indicate an access to the JT1001 controller.

Table 3-6. PDL Receive Pseudo-Code

```
(1)  Function PDLQueueReceiveCommand(PDL, ReceivePacketDescriptor)
(2)      If (ReceiveCommandsAvailable = 0)
(3)          Read RXCMFECN from the Command Status Register (CSR51).                  @
(4)          If (RXCMFECN = 0)
(5)              Return indicating no more receive commands are available.
(6)          Else
(7)              ReceiveCommandsAvailable = the value read from RXCMFECN.
(8)          Endif
(9)      Endif
(10)     TotalLength = 0.
(11)     PDLFragmentIndex = 0
(12)     For (each fragment in the ReceivePacketDescriptor)
(13)         If (the fragment is not in locked memory)
(14)             Call the operating system to lock the memory.
(15)         Endif
(16)         If (the fragment address is a virtual address)
(17)             Call the operating system to convert the virtual address to a list
                     of physical addresses.
```

Table 3-6. PDL Receive Pseudo-Code (Continued)

```
(18)            Endif
(19)            For (each of the virtual fragment's physical addresses)
(20)                Set PDL.FGAD[PDLFragmentIndex] to the physical fragment  address.
(21)                Set PDL.FGLE[PDLFragmentIndex] to the number of bytes in
                        the physical fragment.
(22)                TotalLength = TotalLength + the number of bytes in the physical fragment.
(23)                Move to the next physical fragment in the physical address list.
(24)                Increment PDLFragmentIndex.
(25)            Endfor
(26)        Endfor
(27)        Set PDL.PKLE field to the TotalLength calculated.
(28)        Set PDL.FGCN field to the number of fragments in the ReceivePacketDescriptor.
(29)        Set the desired per packet processing options in the PDL header.
(30)        Enqueue the PDL onto the ReceiveCommandsInProgressQueue.
(31)        Write the MSD of the PDL's physical address to Receive PDL Address  MSD Register @
                (CSR15)
(32)        Write the LSD of the PDL's physical address to Receive PDL Address LSD Register @
                (CSR14)
(33)        Decrement the ReceiveCommandsAvailable count.
(34)        Return Success.
(35)    Endfunction
(36)
(37)    Function PDLProcessReceiveEvent(ReceiveCommandsDone)
(38)        While (ReceiveCommandsDone > 0)
(39)            Dequeue the PDL from the ReceiveCommandsInProgressQueue.
(40)            Get the ReceivePacketDescriptor corresponding to this PDL.
(41)            Update the length fields in the ReceivePacketDescriptor to reflect the lengths
                    returned in the PDL.
(42)            Examine the receive status codes in the PDL receive header and update the
                    ReceivePacketDescriptor as necessary.
(43)            Call the protocol stack to give it the ReceivePacketDescriptor.
(44)            Call the operating system to get another ReceivePacketDescriptor.
(45)            Call PDLQueueReceiveCommand(PDL, ReceivePacketDescriptor).
(46)            If (return code indicates no more receive command were available)
(47)                Free the ReceivePacketDescriptor just allocated.
(48)                Put the PDL back into the PDLAvailable pool.
(49)            Endif
(50)            Decrement ReceiveCommandsDone.
(51)        Endwhile
(52)        Return Success.
(53)    Endfunction
```

The PDL receive pseudo-code uses the following variables to model the state of the JT1001 controller's receiver:

- ReceiveCommandsAvailable — This variable is the minimum number of additional receive commands the JT1001 controller can take at any given point in time. Each time a receive PDL command is given to the JT1001

controller, this count decrements. When it reaches 0, the count is refreshed by reading the RXCMFECN field in the *Command Status Register*.

- ReceiveCommandsDone — This variable is effectively the RXDMDNCN from the *Command Status Register* (CSR51). The RXDMDNCN is also aliased into the *Event Status Register* (CSR32). Each time the *Event Status Register* is read, the RXDMDNCN is added to ReceiveCommandsDone. This determines how many items are taken off the ReceiveCommandsInProgressQueue following an interrupt.
- ReceiveCommandsInProgressQueue — This is a FIFO queue of PDL receive commands issued to the JT1001 controller. This queue preserves the ordering in which the commands were issued to the JT1001 controller. When a PDL receive command is given to the JT1001 controller, the PDL is enqueued on this queue. PDLs are taken off this queue when an RXDMDNIN interrupt occurs.
- ReceivePDLAvailableList — This is a list of PDLs that are available for receiving packets. This pool of receive PDLs is allocated at initialization time. Each time a packet is received using the PDL I/O method, a PDL is removed from this list. PDLs are returned to this list after a PDL receive command has completed and the received packet has been offered to the protocol stack.

The PDLQueueReceiveCommand() performs the following tasks:

- Determines if the JT1001 controller can accept more receive commands.
- Prepares the fragments that constitute the receive buffer for the JT1001 controller. Preparing the fragments involves:
  - — Locking the fragments memory.
  - — Acquiring the physical addresses of the fragments.
- Constructs a pre-receive PDL using the physical addresses and lengths of the fragments.
- Writes the physical address of the PDL to the JT1001 controller's *Receive PDL Address Registers* (CSR14 and CSR15).

PDLProcessReceiveEvent() is called by the interrupt handler when the JT1001 controller has completed the processing for one or more PDL receive commands. ReceiveCommandsDone indicates the number of PDL receive commands that have been completed. For each PDL receive command completed, PDLProcessReceiveEvent performs the following tasks:

- Dequeues the PDL from the ReceiveCommandsInProgressQueue.
- Updates the receive descriptor associated with the PDL.
- Calls the protocol stack and gives it the receive descriptor for the received packet. In this case the protocol owns the receive descriptor and is responsible for freeing it.
- Constructs another a PDL receive command and issues it to the JT1001 controller.

3.6.1 Packet Propulsion Mode Method of Reception

3.6.1.1 PACKET PROPULSION MODE RECEIVE ALGORITHM

The pseudo-code in Table 3-7 demonstrates how to receive a packet using the PDC data transfer method. Lines with an "@" in the right-hand column indicate an access to the JT1001 controller.

Table 3-7. PDC Receive Pseudo-Code

```
(1)  Function PDCQueueReceiveCommand(PDC)
(2)      If (ReceiveCommandsAvailable = 0)
(3)          Read RXCMFECN from the Command Status Register (CSR51).          @
(4)          If (RXCMFECN = 0)
(5)              Return indicating no more receive commands are available.
(6)          Else
(7)              ReceiveCommandsAvailable = the value read from RXCMFECN.
(8)          Endif
(9)      Endif
(10)     Construct the PDC Receive command with PDC length, PDC Buffer ID, and the desired
             setting for RXINRQ.
(11)     Enqueue the PDC on the ReceiveCommandsInProgressQueue.
(12)     Write the command to Receive PDC Register (CSR17).                   @
(13)     Decrement the ReceiveCommandsAvailable count.
(14)     Return Success.
(15) Endfunction
(16)
(17) Function PDCProcessReceiveEvent(ReceiveCommandsDone)
(18)     While (ReceiveCommandsDone > 0)
(19)         Dequeue the PDC from the ReceiveCommandsInProgressQueue.
(20)         ReceiveHeader = the virtual address of the PDC.
(21)         Set the PDC's use count = 0.
(22)         For (each packet in the PDC)
(23)             Get a ReceivePacketDescriptor from the operating system.
(24)             If (no ReceivePacketDescriptors are available)
(25)                 If (PDC's use count is non-zero)
(26)                     Put the PDC on the PDCLoanedToProtocolList.
(27)                 Else
(28)                     Put the PDC on the ReceivePDCAvailableList.
(29)                 Endif
(30)                 Return indicating packets were lost due to lack of HOST resources.
(31)             Endif
(32)             PacketLength = the ReceiveHeader.LEN field.
(33)             Set the packet length in the ReceivePacketDescriptor to PacketLength.
(34)             Set the fragment count in the ReceivePacketDescriptor to 1.
(35)             Set the first fragment virtual address in the ReceivePacketDescriptor
                     to the packet's first RXDATA byte in the PDC; i.e., the
                     CurrentPacketaddress of the first  byte past the packet's receive header in
                     the PDC.
(36)             Examine the receive status codes in the packet's PDC receive
                     header and update the ReceivePacketDescriptor as necessary.
```

Table 3-7. PDC Receive Pseudo-Code (Continued)

```
(37)                Advance the ReceiveHeader to the next packet in the PDC; i.e., the (sum
                        of ReceiveHeader + PacketLength + size of the receive header) rounded
                        up to the next QWORD boundary.
(38)                Call the protocol stack to give it the ReceivePacketDescriptor.
(39)                If (the protocol is not finished with the buffer)
(40)                    Increment the PDC's use count.
(41)                Endif
(42)            Endfor
(43)            If (PDC's use count is not zero)
(44)                Put the PDC on the PDCLoanedToProtocolList.
(45)                Get another PDC from the ReceivePDCAvailableList
(46)            Endif
(47)            If (PDC pointer is not null)
(48)                Call PDCQueueReceiveCommand(PDC) to queue up another receive
                        command.
(49)                If (the return code indicates no more receive commands are available)
(50)                    Put the PDC on the ReceivePDCAvailableList.
(51)                Endif
(52)            Endif
(53)            Decrement ReceiveCommandsDone.
(54)        Endwhile
(55)        Return Success.
(56)    Endfunction
(57)
(58)    Function PDCReceiveDone(ReceivePacketDescriptor)
(59)        Get the PDC that corresponds to the ReceivePacketDescriptor.
(60)        Decrement the PDC's use count.
(61)        Free the ReceivePacketDescriptor.
(62)        If (the PDC's use count is zero)
(63)            Remove the PDC from the PDCLoanedToProtocolList.
(64)            Call PDCQueueReceiveCommand(PDC) to queue up another receive command.
(65)            If (the return code indicates no more receive commands are available)
(66)                Put the PDC on the ReceivePDCAvailableList.
(67)            Endif
(68)        Endif
(69)    Endfunction
```

The PDC receive pseudo-code uses the following variables to model the state of the JT1001 controller's receiver:

- ReceiveCommandsAvailable — This variable is the minimum number of additional receive commands the JT1001 controller can take at any given point in time. Each time a receive PDC command is given to the JT1001 controller, this count decrements. When it reaches 0, the count is refreshed by reading the RXCMFECN field in the *Command Status Register*.
- ReceiveCommandsDone — This variable is effectively the RXDMDNCN from the *Command Status Register* (CSR51). The RXDMDNCN is also

aliased into the *Event Status Register* (CSR32). Each time the *Event Status Register* is read, the RXDMDNCN is added to ReceiveCommandsDone. This determines how many items are taken off the ReceiveCommandsInProgressQueue following an interrupt.

- ReceiveCommandsInProgressQueue — This is a FIFO queue of PDC receive commands issued to the JT1001 controller. This queue preserves the ordering in which the commands were issued to the JT1001 controller. When a PDC receive command is given to the JT1001 controller, the PDC is enqueued on this queue. PDCs are taken off this queue when an RXDMDNIN interrupt occurs.
- ReceivePDCAvailableList — This is a list of PDCs that are available for receiving packets. This pool of receive PDLs is allocated at initialization time. Each time a packet is received using the PDC I/O method, a PDC is removed from this list. PDCs are returned to this list when the protocol stack has finished processing all packets in the PDC and the JT1001 controller can not accommodate any more PDC receive commands.
- PDCLoanedToProcotolList — This is a list of PDCs whose use counts were not zero when PDCProcessReceiveEvent finished processing the PDC. This occurs if one or more packets contained in the PDC are still in use by the protocol stack. The protocol stack calls the PDCReceiveDone() function.

The PDCQueueReceiveCommand() performs the following tasks:

- Determines if the JT1001 controller can accept more receive commands.
- Enqueues the PDC to the ReceiveCommandsInProgressQueue.
- Writes the PDC's buffer ID and length to the JT1001 controller's *Receive PDC Register* (CSR17).

PDCProcessReceiveEvent() is called by the interrupt handler when the JT1001 controller has completed the processing for one or more PDC receive commands. ReceiveCommandsDone indicates the number of PDC receive commands that have been completed. For each PDC receive command completed, PDCProcessReceiveEvent performs the following tasks:

- Dequeues the PDC from the ReceiveCommandsInProgressQueue.
- Parses the PDC packet for the beginning of each packet received into the PDC. For each packet in the PDC, it does the following:
  - — Allocates and initializes the receive descriptor to point to the data in the PDC and include the receive status of the packet.
  - — Calls the protocol stack to give it the receive packet descriptor. In this case, the protocol stack either completely processes the packet before returning, or calls the PDCReceiveDone() at some later point in time to indicate it has finished processing the packet.
- Maintains a use count for each PDC to keep track of how many packets within the PDC are in use by the protocol stack. PDCs whose use count is non-zero are added to the PDCLoanedToProcotolList.
- Calls PDCQueueReceiveCommand() to issue another PDC receive command to the JT1001 controller.

PDCReceiveDone() is called by the protocol stack to indicate it is finished processing a received packet. The use count for the corresponding PDC is decremented. If the PDC use count is zero, PDCReceiveDone() attempts to issue another PDC receive command to the JT1001 controller.

3.7 INTERRUPT PROCESSING

3.7.1 Event Status Register

The *Event Status Register* acts as an accumulator of JT1001 controller events. The JT1001 controller tracks the following categories of events in the *Event Status Register:*

- Receive events.
- Transmit events.
- FIFO watermark events.
- Timer events.

As events occur, their corresponding event bits get set in the *Event Status Register*. Event bits remain set until the register is read by HOST software. When the *Event Status Register* is read, the JT1001 controller returns the current value of the register to HOST software and then clears the register. Only the bits that are read are cleared. In other words, if HOST software does a byte access to the second byte in the *Event Status Register*, only that byte is cleared.

The *Event Status Register* has two special attributes. First, the high order byte of the register is actually an alias for the RXDMDNCN in the *Command Status Register*. When this field is read, the JT1001 controller automatically clears it in both the *Event Status Register* and the *Command Status Register*.

Second, a read of this register also causes the INENMS bit in the *Interrupt Mask Register* to be cleared if an interrupt is pending (i.e., the JT1001 controller's interrupt line is active). This has the effect of disabling the JT1001 controller's ability to generate further interrupts. It is the responsibility of HOST software to re-enable the JT1001 controller's ability to generate an interrupt by setting the INENMS bit in the *Interrupt Mask Register*.

3.7.2 Interrupt Mask Register

The *Interrupt Mask Register* governs the JT1001 controller's ability to generate interrupts on the PCI bus. The INENMS bit in the *Interrupt Mask Register* is the JT1001 controller's master enable/disable switch for interrupt generation. If INENMS is set, the JT1001 controller has the capability to generate an interrupt. If INENMS is clear, the JT1001 controller can not generate an interrupt.

All other bits in the *Interrupt Mask Register* determine which events in the *Event Status Register* generate an interrupt. For each event bit in the *Event Status Register*, there is a corresponding interrupt mask bit in the *Interrupt Mask Register*. If the INENMS bit is set and the event bit's corresponding mask bit is set in the *Interrupt Mask Register*, the JT1001 controller generates an interrupt whenever the event bit gets set in the *Event Status Register*. Note that the

Interrupt Mask Register does not prevent bits from being set in the *Event Status Register,* it merely determines which events cause the JT1001 controller to generate an interrupt.

3.8 INTERRUPT HANDLER

The pseudo-code in Table 3-8 demonstrates a typical interrupt handler for processing JT1001 controller interrupts.

Table 3-8. Interrupt Handler Pseudo-Code

```
(1)  Function InterruptHandler(PDC)
(2)      Read the Interrupt Mask Register (CSR30) and save it in InterruptMask.        @
(3)      If (the INENMS bit is not set)
(4)          Return indicating the CONTROLLER did not generate the interrupt.
(5)      Endif
(6)      Read the Event Status Register (CSR32). NOTE: The act of reading CSR32 will clear the   @
             INENMS bit in CSR30 if the CONTROLLER's interrupt line is high when the read
             occurs. This has the effect of disabling the CONTROLLER's ability to generate
             interrupts.
(7)      EventStatus = EventStatus OR with the value just read from CSR30.
(8)      If (using PDC or PDL method to receive packets)
(9)          ReceiveCommandsDone = RXDMDNCN from EventStatus +
                 ReceiveCommandsDone.
(10)     Endif
(11)     If ((EventStatus AND InterruptMask) is zero)
(12)         Return indicating the CONTROLLER did not generate the interrupt.
(13)     Endif
(14)     If (required by the operating system)
(15)         Issue an EOI to the interrupt CONTROLLER.
(16)         Enable interrupts at the CPU.
(17)     Endif
(18)     While (EventStatus AND InterruptMask) is not zero)
(19)         If (using PDC method for receiving packets)
(20)             Call PDCProcessReceiveEvent(ReceiveCommandsDone).                @
                     See Table 3-7
(21)         Else If (using PDL method for receiving packets)
(22)             Call PDLProcessReceiveEvent(ReceiveCommandsDone).                @
                     See Table 3-6
(23)         Else using PIO method for receives
(24)             If ((EventStatus AND RXIN) is not zero)
(25)                 Call PIOProcessReceiveEvent(). See Table 3-5                 @
(26)             Endif
(27)         Endif
(28)         If ((EventStatus AND TXDMDNIN) is not zero)
(29)             TransmitCommandsDone = Read TXDMDNCN from the                    @
                     Command Status Register (CSR51).
(30)             While (TransmitCommandsDone > 0)
(31)                 Dequeue from the TransmitCommandsInProgressQueue.
(32)                 If (the item dequeued was a PDC)
```

Table 3-8. Interrupt Handler Pseudo-Code (Continued)

```
(33)                    Call PDCTransmitDMADoneEvent(PDC).
                            See Table 3-4
(34)                Else the item dequeued was a PDL
(35)                    Call PDLTransmitDMADoneEvent(PDL).
                            See Table 3-4.
(36)                Endif
(37)                Decrement the TransmitCommandsDone count.
(38)            Endwhile
(39)        Endif
(40)        Read the Event Status Register (CSR32) and save the result in          @
                EventStatus.
(41)        If (using PDC or PDL method to receive packets)
(42)            ReceiveCommandsDone = RXDMDNCN from EventStatus +
                    ReceiveCommandsDone.
(43)        Endif
(44)    Endwhile
(45)    Enable CONTROLLER interrupt by setting INENMS in the Interrupt Mask    @
            Register (CSR30).
(46)    Return Success.
(47) Endfunction
```

The interrupt handler pseudo-code above is designed to operate with the initialization, transmit, and received pseudo defined earlier. It is intended to demonstrate the fundamental organization of the interrupt handler. It is not necessarily the optimal way to organize the interrupt handler.

The interrupt handler pseudo-code uses the following variables for interrupt processing:

- InterruptMask — This variable is the value read from the *Interrupt Mask Register* (CSR30). It is used to determine whether the JT1001 controller's master interrupt enable bit INENMS is set. InterruptMask is also used to mask values read from the *Event Status Register* (CSR32).
- EventStatus — This variable accumulates events read from the *Event Status Register* (CSR30). EventStatus is bitwise ORed with the initial read of the *Event Status Register* in InterruptHandler(). EventStatus is set to the value read from the *Event Status Register* each time InterruptHandler() iterates through the event processing loop.
- ReceiveCommandsDone — This variable is the RXDMDNCN from the *Command Status Register* (CSR51). It indicates the number of PDC or PDL commands that the JT1001 controller has completed processing and need to be processed by the driver.
- TransmitCommandsDone — This variable is the TXDMDNCN from the *Command Status Register* (CSR51). Each time a TXDMDNIN occurs, the TXDMDNCN field is read and its value is saved in TransmitCommandsDone. This determines how many items are taken off the TransmitCommandsInProgressQueue following a TXDMDNIN.
- TransmitCommandsInProgressQueue — This is a FIFO queue of PDL and PDC commands issued to the JT1001 controller. This queue preserves the

sequence and type of transmit commands issued to the JT1001 controller. When a transmit PDL or PDC command is given to the JT1001 controller, the PDL or PDC is enqueued on this queue. Items are taken off this queue when a TXDMDNIN interrupt occurs.

InterruptHandler() is called by the operating system when the interrupt occurs on the interrupt line for which the handler is registered. The interrupt handler first determines whether the JT1001 controller generated the interrupt. To determine this, it reads the *Interrupt Mask Register* (CSR32) and checks the INENMS bit. If INENMS is clear, the JT1001 controller did not generate the interrupt so the interrupt handler exits. If the INENMS is set, HOST software must then read the *Event Status Register* and bitwise AND the value read with the value read from the *Interrupt Mask Register*. If the result is of this operation is non-zero, the JT1001 controller generated the interrupt, otherwise some other device generated the interrupt.

It is important to note that the value read from the *Event Status Register,* when determining whether the JT1001 controller generated the interrupt, must be saved even if the JT1001 controller did not generate the interrupt. This is necessary because of the clear after read nature of the *Event Status Register*. The value read from the *Event Status Register* is bitwise ORed with the EventStatus variable, even when the JT1001 controller did not generate the interrupt. This prevents the loss of JT1001 controller events for which the JT1001 controller is not configured to generate an interrupt. For similar reasons, if the driver is using PDL or PDC mode to receive frames, the RXDMDNCN read from the *Event Status Register* is added to ReceiveCommandsDone.

Once the InterruptHandler() has determined the JT1001 controller generated an interrupt, it proceeds to the event processing loop. For each iteration of the loop, the appropriate event processing occurs for each bit set in EventStatus. At the bottom of the loop, the *Event Status Register* is read again and EventStatus and ReceiveCommandsDone are updated. The InterruptHandler() continues looping until EventStatus and ReceiveCommandsDone are 0.

Note that this implementation of the interrupt handler checks to see the type of receive I/O method being used for receive, because the driver implements all three modes. Typically, a driver implements a single receive I/O method that is optimal for the target operating system. In this case, the check to determine the receive I/O method being used is unnecessary.

3.9 VLAN SUPPORT

The JT1001 controller provides the following VLAN IEEE 802.1Q support with the following functionality:

- VLAN tag insertion.
- VLAN tag removal.
- VLAN tag packet filtering.

A VLAN tag is 4 bytes long and consists of a 2-byte protocol ID field followed by a 2-byte tag control information (TCI) field. The TCI field is divided into a 3-bit

user priority field, a 1-bit canonical format identifier (CFI) field, and a 12-bit VLAN Identifier (VLID) field. VLAN tag headers are located at bytes offset 12 – 15 in a packet's MAC header (i.e., between the source address field and length/type field). The protocol ID field is used by the JT1001 controller for VLAN tag insertion and filtering. Its value can be configured by HOST software using the *VLAN Tag Protocol ID Register*. The JT1001 controller also provides a 16-entry table for TCIs. The VLAN Tag TCI table is used by the JT1001 controller for VLAN tag insertion and filtering. HOST software adds and deletes entries in the table using the *VLAN Tag TCI Table Register.*

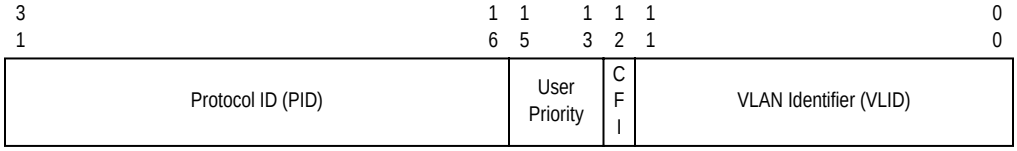


Figure 3-5. VLAN Header Format

The VLEN bit in *Mode Register – 1* is the master enable/disable bit for the JT1001 controller's VLAN support. By default the VLEN bit is clear, meaning the JT1001 controller's VLAN support is disabled. When the VLEN bit is set, the JT1001 controller's VLAN support is enabled and HOST software can then independently configure and enable the three functions described above.

The JT1001 controller's VLAN tag insertion function can operate in two different modes: global and per-packet. In global mode, the JT1001 controller inserts a VLAN tag header in all packets transmitted. The JT1001 controller constructs and inserts a VLAN tag using the value in the *VLAN Tag Protocol ID Register* and the value in the first (0) entry of the VLAN Tag TCI Table. HOST software enables the global mode of VLAN tag insertion by setting the VLEN and VLISGB bit in *Mode Register – 1*.

Per-packet VLAN tag insertion gives HOST software the capability to request the VLAN tag insertion on a per-packet basis. First, the VLEN bit must be set. HOST software then sets the VLIS bit in the packet's transmit header to indicate the VLAN tag header is to be inserted. When VLIS is set, HOST software must also set the VLTBIX field in the packet's transmit header. The VLTBIX field is an index into the VLAN Tag TCI Table. The JT1001 controller constructs and inserts the VLAN tag header using the value in the *VLAN Tag Protocol ID Register* and the TCI information using the VLAN Tag TCI Table entry specified by VLTBIX.

HOST software can mix the global and per-packet tag insertion modes by setting the VLISGB in *Mode Register – 1* and then setting the VLIS bit in the transmit header of selected packets. In this case, the VLIS prevails and the JT1001 controller constructs and inserts a VLAN tag header using the per-packet method.

The JT1001 controller's VLAN tag removal support can be enabled by setting the VLEN and VLRMID bits in *Mode Register – 1*. When enabled, the JT1001 controller parses inbound packets for a VLAN tag header. If a VLAN tag header is found, it is removed from the packet prior to being put into the RX FIFORX

FIFO. Consequently, the VLAN tag is never seen by HOST software. When the JT1001 controller removes a VLAN tag header from a packet, the CRC is always removed too, regardless of the state of the PACREN bit in *Mode Register – 1*. The CRC is removed because it is no longer valid since the VLAN tag is included in the CRC calculation.

Finally, the JT1001 controller can filter inbound packets that contain a VLAN tag header. The VLAN tag filter affects received packets that contain a VLAN header. When configured by HOST software, the VLAN tag filter forwards packets containing a VLAN header whose protocol ID matches the value set in the *VLAN Tag Protocol ID Register* and whose VLID field matches the VLID of an entry the VLAN Tag Table. This feature is enabled by setting the VLTBEN bit in *Mode Register – 1*. When the VLAN tag filter is enabled, the JT1001 controller provides additional receive status information in the packet's PIO/PDL/PDC receive header. In particular, the JT1001 controller sets the VLHT bit in the receive header of packets that contain a VLAN tag header that passed the VLAN tag filter. The JT1001 controller also puts the index of the matching VLAN Tag TCI Table entry in VLTBIX field of the receive header.

3.10 TCP/IP CHECKSUM SUPPORT

The JT1001 provides advanced capabilities for processing TCP/IP checksums. The JT1001 controller can calculate and insert IP, TCP, and UDP checksums during packet transmission and verify IP, TCP, and UDP checksums during packet reception. The JT1001 controller overlaps the checksum processing with packet transmission and reception. The JT1001 controller's TCP/IP checksum capabilities increase system performance by overlapping checksum processing with the packet transmission/reception and by relieving the HOST CPU of the checksum calculation and verification tasks.

The JT1001 controller's ability to calculate and insert a checksum to a packet during packet transmission can be enabled independently for IP, TCP, and UDP. Setting the TXIPCKEN bit in *Mode Register – 2* enables IP checksum insertion. When set, the JT1001 controller inserts the IP checksum into all outbound packets that contain an IP version 4 header. Alternatively, HOST software can choose to insert the checksum on a per-packet basis by clearing the TXIPCKEN bit and setting the IPCKIS bit in the packet's PIO/PDL/PDC transmit header. The TXTPCKEN and TXUPCKEN bits in *Mode Register – 2* enable the TCP and UDP checksum calculation and insertion on a global basis. The TPCKIS and UPCKIS bits in the packet's PIO/PDL/PDC transmit header cause the TCP and UDP checksums to be calculated and inserted on a per-packet basis.

There are three bits in *Mode Register – 2* that enable/disable the JT1001 controller's ability to verify TCP/IP checksums in received packets. Setting the RXIPCKEN bit in *Mode Register – 2* causes the JT1001 controller to verify the IP checksum for all inbound packets containing an IP header. If the checksum fails, the JT1001 controller sets the IPCKER status bit in the packet's PIO/PDL/PDC receive header. If the checksum is valid or the packet does not contain an IP header, the IPCKER bit will be clear in the receive header. The JT1001 controller handles checksums for TCP and UDP in a similar manner. The RXTPCKEN and RXUPCKEN bits in *Mode Register – 2* enable the TCP

and UDP checksum verification. The TPCKER and UPCKER bits in the packet's PIO/PDL/PDC receive header will be set if a TCP or UDP checksum error occurs.

Finally, the action the JT1001 controller takes for packets containing TCP/IP checksum errors is governed by the PACKEREN bit in *Mode Register – 2*. When PACKEREN is set, received packets containing TCP/IP checksum errors are passed to HOST software. When clear, the packets are discarded by the JT1001 controller.

3.10.1 EEPROM Support

The JT1001 provides a 93C46 compatible EEPROM interface. EEPROM is used as a convenient nonvolatile store of JT1001 controller parameters. In particular, EEPROM is used to store the JT1001 controller's Universally Administered Address (UAA), PCI configuration space defaults, and overrides for some CSR's default values. EEPROM can also be used by HOST software as a nonvolatile store for software configuration parameters. Typically, EEPROM is accessed following a JT1001 controller reset or by HOST diagnostic software.

Following a JT1001 controller hard or soft reset, the JT1001 controller reads EEPROM and overwrites the CSRs. See Figure 7-1 to determine which CSRs are overwritten following a reset. When reloading CSRs from EEPROM, the JT1001 controller calculates a 32-bit checksum and compares the checksum against the checksum value stored in EEPROM. If an incorrect checksum is obtained, the JT1001 controller restores CSRs overwritten by EEPROM to their default values.

HOST software can access EEPROM using the *EEPROM* and *EEPROM Data Registers* on an individual DWORD basis. Although EEPROM is organized internally as 16-bit words, the JT1001 controller presents the EEPROM contents as 32-bit DWORDs to HOST software. The JT1001 controller accomplishes this by combining two EEPROM 16-bit words into a 32-bit big endian DWORD.

Since the JT1001 controller can operate without EEPROM, HOST software must determine if EEPROM is present prior to accessing it. If EEPROM is present, the JT1001 controller sets the EEPMPN bit in the *EEPROM Register*. To read a value from EEPROM, HOST software sets EEAD in the *EEPROM Register* to the index of the EEPROM DWORD to be read. It then sets EERDCM and EESL bits in the *EEPROM Register* to initiate the read command. HOST software must then poll the EESL bit to await the completion of the EEPROM read command. The JT1001 controller clears the EESL bit after completing the read command. HOST software can then read the value from the *EEPROM Data Register.*

To write a value to EEPROM, HOST software first writes the value to be written to EEPROM to the *EEPROM Data Register*. Next, HOST software sets EEAD in the *EEPROM Register* to the index of the EEPROM DWORD to be written. It then sets EEWTCM and EESL bits in the *EEPROM Register* to initiate the write command. HOST software must then poll the EESL bit to await the completion of the EEPROM write command. The JT1001 controller clears the EESL bit after completing the write command. If HOST software writes a value in EEPROM, it must also recalculate and write the new checksum value to EEPROM.

3.11 EXPANSION ROM SUPPORT

The JT1001 provides an expansion ROM interface. If present, the expansion ROM contains an executable image that is invoked by the HOST system's BIOS during the boot process. An expansion ROM attached to the JT1001 typically contains an executable image that allows the network connection to be the boot device (i.e., the device from which the operating system is loaded). The JT1001 expansion ROM interface supports EPROM and flash devices.

HOST BIOS detects the presence and size of the expansion ROM using the *Expansion ROM Base Address Register* at offset 30h in the PCI Configuration space. If the expansion ROM is present, HOST BIOS maps the expansion ROM into the HOST's memory address space by writing at the base address to the *Expansion ROM Base Address Register*. Due to the slow access times for expansion ROM devices, BIOS shadows (i.e., copies) the expansion ROM image into system RAM. When BIOS executes the expansion ROM image, it executes the expansion ROM image copied to RAM.

Once BIOS has mapped expansion ROM into HOST memory address space, HOST software can read the expansion ROM just as it reads any other memory location. HOST software can read the expansion ROM using 8-, 16-, and 32-bit memory accesses. Due to the slow access times of most flash devices, however, HOST software should avoid 32-bit read accesses after system initialization time to ensure the PCI 16- and 8-clock bus holding rules are not violated. The PCI specification allows the bus holding rules to be violated during system initialization so expansion ROMs can be shadowed to system RAM using 32-bit accesses.

If the expansion ROM is a flash device, HOST software can also perform memory writes to the device. HOST software can determine whether a flash device is present by checking the FLPN bit in the *EEPROM Register*. To enable writing to the flash device, HOST software must first set the FLWTEN bit in the *EEPROM Register*. HOST software then writes to the flash device using 8-bit accesses. Due to the slow access times of most flash devices, HOST software should avoid 16- and 32-bit write accesses to ensure the PCI 16- and 8-clock bus holding rules are not violated. If either the FLPN or FLWTEN bits are clear, write accesses to the expansion ROM do not change the contents of the expansion ROM.

3.11.1 Magic Packet Wake Up

The JT1001 supports wake up via Magic Packet technology. Magic Packet technology, developed by Advanced Micro Devices, allows a computer system in a low or no power state to be restored to a full power state remotely via network. This is accomplished by sending a packet containing a specific data sequence. This is referred to as a Magic Packet data sequence. The JT1001 requires the power to be supplied to the JT1001 controller via the PCI bus, however, all other components in the computer system can be in a low or no power state.

In order for a packet to be considered a Magic Packet data sequence, it must meet the following criteria:

- It must contain a valid 14-byte MAC header; i.e., 6-byte destination address, 6-byte source address, and 2-byte length/type field. The destination address can be a unicast, multicast, or broadcast address.
- The packet must be have a valid CRC and be at least the minimum frame size in length.
- The LLC data portion of the packet must contain a 6-byte preamble 0xFF, followed by 16 repetitions of the JT1001 controller's MAC address. The preamble and MAC address repetitions must be contiguous. The sequence itself can begin at any offset within the LLC data.

It is necessary to permit Magic Packet data sequences to contain a multicast or broadcast destination address to ensure the computer is reachable through routers. Some protocols cause routers to discard the unicast MAC address of a computer from their routing tables when the computer is powered off. In such instances, a Magic Packet data sequence containing a unicast destination address can not be routed to the computer. A Magic Packet data sequence containing a broadcast or multicast address, however, can always be routed to a computer and, therefore, allow the computer to be successfully awakened.

The JT1001 controller is put into Magic Packet mode by setting the MGPKEN bit in *Mode Register – 1*. When a Magic Packet data sequence is received, the RXMGPKIN bit is set in the *Event Status Register*. If the RXMGPKMS and INENMS bit are set in the *Interrupt Mask Register*, an interrupt on the PCI bus is generated.

The reception of a Magic Packet data sequence can also cause the system to wake up via the power management event ($\overline{\text{PME}}$) pin. In a computer that supports PCI power management, assertion of the $\overline{\text{PME}}$ causes the system to return to a fully powered state. Although the $\overline{\text{PME}}$ pin is defined by the PCI power management specification, the output of this pin can be routed to the wake up circuitry of a computer that does not support PCI power management. The following section discusses how to configure the JT1001 controller such that a Magic Packet data sequence results in the $\overline{\text{PME}}$ pin being asserted.

3.12 PCI POWER MANAGEMENT

The JT1001 complies with the PCI Power Management Interface Specification, Rev. 1.0. This specification defines a set of PCI configuration space registers used to query a PCI device's power management capabilities, and query and set its power management state. Additionally, a $\overline{\text{PME}}$ pin is defined for signaling wake up events to the computer system. System-level software uses these interfaces to manage the power state of the PCI devices. Refer to the PCI Power Management Specification, Rev. 1.0, for a detailed description of these registers and pin.

The JT1001 supports power management in the following manner:

- Supports device power states D0, $D3_{hot}$, and $D3_{cold}$.
- No explicit action is taken to power down blocks within the JT1001 controller. Power savings occur because the PCI block and MAC will not put data into the Tx and RX FIFOs during the $D3_{hot}$ state.

- $\overline{\text{PME}}$ assertion can occur due to Magic Packet data sequence frame recognition during either the D0 or $D3_{hot}$ power states.
- PCI Configuration space accesses are enabled while in the D0 and $D3_{hot}$ power state.
- Interrupts, PCI memory transactions, and PCI I/O transactions are disabled in $D3_{hot}$.
- The Power Management Register Block is implemented in the PCI configuration space.

The PME_En bit in the *Power Management Control/Status Register* (PMCSR) governs whether or not a Magic Packet event results in the $\overline{\text{PME}}$ pin being asserted. The MGPKEN enable bit in CSR 00 enables the Magic Packet detection logic in the MAC. For a Magic Packet data sequence to be detected and cause $\overline{\text{PME}}$ to be asserted, both the MGPKEN bit in CSR 00 and the PME_En bit in PMCSR must be set. When these bits are set and a Magic Packet data sequence is received, the $\overline{\text{PME}}$ pin is asserted and the JT1001 controller resets itself as if a PCI $\overline{\text{RST}}$ had occurred, with one exception: the PME context is preserved across the reset. The PME context is defined as the state of all bits in the *Power Management Control/Status Register* and the state of the $\overline{\text{PME}}$ pin. The $\overline{\text{PME}}$ will remain asserted until either the PME_Status bit or the PME_En bit is cleared by software.

3.13 PRE-FETCHING

Although memory mapped I/O is supported by the JT1001, I/O pre-fetching is not. Consequently, memory areas allocated to JT1001 controller I/O must not be cached.

Section 4
PCI Configuration Registers

00h	Device ID (0001h)		Vendor ID (1308h)	
04h	Status (00B0h)		Command (0000h)	
08h	Class Code (020000h)			Revision ID (00h)
0Ch	BIST (00h)	Header Type (00h)	Latency Timer (0Dh)	Cache Line Size (00h)
10h	Base Address Register 0: 32-bit I/O base address (00000001h)			
14h	Base Address Register 1: 64-bit non-prefetchable memory base address – LSD (00000004h)			
18h	Base Address Register 2: 64-bit non-prefetchable memory base address – MSD (00000000h)			
1Ch	Base Address Register 3: Not used by JT1001. (00000000h)			
20h	Base Address Register 4: Not used by JT1001. (00000000h)			
24h	Base Address Register 5: Not used by JT1001. (00000000h)			
28h	Cardbus CIS Pointer: Not used by JT1001. (00000000h)			
2Ch	Subsystem ID (0001h)		Subsystem Vendor ID (1308h)	
30h	Expansion ROM Base Address (00000000h)			
34h	Reserved (0000000h)			Capabilities Ptr (44h)
38h	Reserved (00000000)			
3Ch	Max_Lat (00h)	Min_Gnt (00h)	Interrupt Pin (01h)	Interrupt Line (00h)
40h	Reserved (0000h)		Retry Timeout (00h)	TRDY Timeout (00h)
44h	Power Management Capabilities Register (4801h)		Next Item Ptr (00h)	Power Mgmt Cap. ID (01h)
48h	Reserved (0000h)		Power Management Control/Status Register (0000h)	
4Ch – FFh	Reserved (00000000h)			

Loaded from EEPROM

Device Dependent Region

Figure 4-1. PCI Configuration Space Register Map

In Figure 4-1, default values following PCI $\overline{\text{RST}}$ appear in parenthesis. The default values for registers loadable from EEPROM become overridden by values specified in EEPROM when EEPROM is present and functioning. Refer to the PCI 2.1 Specification for a detailed description of the PCI configuration registers. Refer to the PCI Bus Power Management Interface Specification, Rev, 1.0, for a detailed description of the PCI configuration registers specific for power management.

Section 5
Command and Status Registers

This section presents the details of the register interface to the JT1001. The information is arranged in tabular form with the following format:

Bit Field	Type	E^2	Mnemonic	Default Value	Description

The individual columns of the table have the following significance:

Bit Field — Indicates the start and end bit position of a field of bits. The most significant bit position is listed first, followed by a colon character (":") and the least significant bit position of the field within the register. For single bit fields, the position number is listed without the ensuing colon and end bit identifier. For example, 20:17 identifies a bit field that is 4 bits wide. The most significant bit is bit 20, and the least significant bit is bit 17. Bit 20 represents a value of 2^3 and bit 17 represents a value of 2^0.

Type — The type field can contain a series of single letter identifiers that denote the behavioral attributes of specific bit fields within a register. The identifiers are concatenated together to denote the attributes that apply to the given bit field. Acceptable type designators are:

- R — Read.
- W — Write.
- A — Auto Clear. (Auto Clear implies the field is readable.)
- C — Clear after read.

E^2 — This column is used to denote if the field's initial value is to be obtained from the serial EEPROM. If a field's initial value is obtained from EEPROM, a check mark (√) is placed in this column. Otherwise, an x mark (x) is placed in the column.

Mnemonic — Values in this column provide symbolic names for the corresponding field. The mnemonics are constructed to aid in the pronunciation of the field's name. Generally, mnemonics are two characters in length and are concatenated to form a single symbol. For example, the mnemonic BFAD is comprised of the two sub-mnemonics BF which represents the word buffer and AD which represents the word address. Together, they symbolize a buffer address. Each mnemonic used in this document is listed in the Glossary, Section 8.

Default Value — The Default Value column denotes the value the register will assume once the JT1001 controller has been powered up and placed into its start state. Fields that are initialized from EEPROM also have a default value. In these cases, the default value is applied prior to EEPROM being read. This method provides a means for the chip to initialize even if the EEPROM should fail, or if an EEPROM is not desired.

Description — The Description column provides a brief explanation of the field and its usage.

One important aspect of the register interface is that it is inherently a 32-bit interface. This is due to the PCI's use of a 32-bit I/O path despite its support for 64-bit data and address paths for memory cycles. The net result of this design attribute of the PCI bus is that 64-bit address registers implemented in the JT1001 controller must be accessed with two 32-bit I/O cycles. To avoid race conditions when writing to 64-bit registers, a policy is adopted by the JT1001 controller whereby it only examines the contents of a 64-bit register once the least significant DWORD (LSD) is written. To help clarify the policy, consider the case where a 64-bit register is maintained by the JT1001 controller and is updated by HOST software. The HOST begins by writing the MSD register. After the MSD is written, the HOST writes the LSD. This policy brings about two noteworthy behaviors. First, it eliminates the potential for a race condition between the HOST and the JT1001 controller. Second, it provides a means by which 64-bit registers can be updated with single 32-bit writes. For example, when initializing the PDC Buffer Address Table (which requires 64-bit physical addresses for PDC data buffers), HOST software can write the contents of the *Transmit PDC Buffer Address MSD Register* prior to initializing the table. Subsequently, the HOST software can consecutively write the LSDs of the PDC buffer addresses to the register without further manipulation of the MSD register. This technique works because the MSDs of the addresses of PDC data buffers are almost certainly the same for all PDC data buffers allocated by HOST software. Once this MSD is written to the *Transmit PDC Buffer Address MSD Register,* it is read by the JT1001 controller *each* time the LSD register is written.

CSR 00 MODE REGISTER – 1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
LNCKEN	USPIMD1	USPIMD0	VLISGB	VLRMID	VLTBEN	VLEN	**SERECL**	RXFLCTEN	MGMCBCEN	MGPKEN	DBMDEN	TXCREN	PAERPKEN	PACREN	**SERECL**	LGPKEN	UCEN	POEN	BCEN	MCEN	RXEN	TXEN	**SERECL**	RMPPEN	TXPPEN	GMSTPOEN	RXTRPR	TXFLCTEN	DEBTOD	SWRE	**SERECL**

Bit Field	Type	E^2	Mnemonic	Default Value	Description
0	W	x	SERECL	0	*Set/Reset Control.* Set/reset control bit for bits[7:1].
1	WA	x	SWRE	0	*Soft Reset.* When set, the JT1001 controller resets all internal hardware with the exception of the PCI configuration registers. This bit remains set for the duration of the reset. Upon completion of the reset, the JT1001 controller automatically clears this bit. HOST software can poll this bit to determine when the reset has completed.
2	RW	√	DEBTOD	0	*Descriptor Byte Ordering.* 0 = Little Endian, 1 = Big Endian
3	RW	√	TXFLCTEN	1	*Transmit Flow Control Enable.* When set, the JT1001 controller automatically constructs and transmits a PAUSE frame when the RX FIFO hits the high and low watermarks specified in the *Flow Control Watermark Register.* When clear, the JT1001 controller does not automatically construct and transmit PAUSE frames.
4	RW	√	RXTRPR	0	*Receive/Transmit Priority.* Bus arbitration priority between receive and transmit. When set, the JT1001 controller uses a round-robin arbitration scheme between receive and transmit (equal priority). When reset, receive has an 8:1 priority over transmit.
5	RW	√	GMSTPOEN	0	*G/MII Status Polling Enable.* When set, the JT1001 controller periodically queries the PHY to determine if a status change has occurred. If a status change has occurred, the JT1001 controller sets the PHLASTIN bit in the *Event Status Register.* When clear, the JT1001 controller does not query the PHY for a status change. HOST software can perform this operation manually via the *G/MII PHY Access Register.*
6	RW	√	TXPPEN	1	*Transmit Packet Pad Enable.* When set, the JT1001 controller pads a transmit packet to the minimum frame size. The minimum frame size for ethernet is 60 bytes, excluding the CRC and VLAN tag field.

Bit Field	Type	E^2	Mnemonic	Default Value	Description
7	RW	√	RMPPEN	1	*Remove Packet Pad Enable.* When set, the JT1001 controller strips pad bytes in a received packet that contains a length field in the MAC header that has a value less than 46 decimal. Pad bytes are defined as bytes following the end of the LLC data field but before the CRC. Since stripping the pad bytes renders the CRC invalid, it is also stripped (regardless of the state of PACREN). When the RMPPEN bit is clear, the JT1001 controller does not strip pad bytes in received packets. The JT1001 controller does not strip pad bytes from packets that contain a type in the MAC header instead of a length. A MAC header has a type field if the value in the length/type field is greater than 1536 bytes.
8	W	x	SERECL	0	*Set/Reset Control.* Set/reset control bit for bits[15:9].
9	RW	x	TXEN	0	*Transmitter Enable.* When set, the JT1001 controller can perform transmits. When reset, the JT1001 controller will not perform transmits. If a packet transmission is in progress, the JT1001 controller will complete it and stop.
10	RW	x	RXEN	0	*Receiver Enable.* When set, the JT1001 controller can perform receives. When reset, the JT1001 controller will not perform receives. If a packet reception is in progress, the JT1001 controller will complete it and stop.
11	RW	x	MCEN	0	*Multicast Enable.* When set, the JT1001 controller will accept a packet with a multicast destination address that matches in the *Multicast Hash Table Register.*
12	RW	x	BCEN	0	*Broadcast Enable.* When set, the JT1001 controller will accept a packet with a broadcast destination address.
13	RW	x	POEN	0	*Promiscuous Mode Enable.* When set, all packets will pass the JT1001 controller's destination address filter, regardless of the settings of UCEN, MCEN, and BCEN. Packets, however, are still subject to the JT1001 controller's other reception filters. To cause the JT1001 controller to accept all packets without modification, HOST software must set the POEN, PAERPKEN, and PACREN bits in *Mode Register – 1*, and clear the VLTBEN, VLRMID, and RMPPEN bits in *Mode Register – 1.*
14	RW	x	UCEN	1	*Unicast Mode Enable.* When set, the JT1001 controller will accept frames in which the destination address matches the JT1001 controller's unicast address; i.e., the address set in the LAN Physical Address Register (CSR 42). When reset, frames with the destination addresses matching the station's unicast address will **not** be accepted.

Bit Field	Type	E^2	Mnemonic	Default Value	Description
15	RW	√	LGPKEN	0	*Large Packet Enable.* This bit determines how the JT1001 controller processes packets that exceed the maximum packet size. When the LGPKEN bit set, the JT1001 controller receives the packets that exceed the maximum packet size into the RX FIFO and sets the LGPK bit in the receive header. The Large Packet Count is incremented. When the LGPKEN bit is clear, the JT1001 controller treats packets that exceed the maximum packet size as an errored packet. If the PAERPKEN bit is clear, the JT1001 controller discards the packet. If the PAERPKEN bit is set, the JT1001 controller receives the packet into the FIFO and sets the LGPK and ERROR bits in the receive header. Regardless of the state of PAERPKEN, the JT1001 controller increments the Large Packet Count and Errored Packet Count. The maximum packet size used by the JT1001 controller varies depending on the state of the VLEN bit. If VLEN is clear, the maximum packets size is 1518 bytes (including CRC). If VLEN is set, the maximum packet size is increased to 1522 bytes to allow for the 4-byte VLAN header.
16	RW	x	SERECL	0	*Set/Reset Control.* Set/reset control bit for bits[23:17].
17	RW	x	PACREN	0	*Pass CRC Enable.* When set, this bit indicates to the JT1001 controller that the HOST software wishes to receive each frame's CRC field. On reception, the only time this bit is defined, the inbound frame's CRC field is transferred to receive buffers as it appears on the wire. Receive buffers are expected to be sufficiently large to accommodate the CRC. If not, a HOST buffer overflow error occurs. Regardless of the state of PACREN, the JT1001 controller does not pass the CRC of packets from which it has removed the packet padding or a VLAN header. Since these fields are included in the CRC calculation, the CRC is no longer valid after the JT1001 controller has removed them. Therefore, the JT1001 controller does not pass the CRC in this case.
18	RW	x	PAERPKEN	0	*Pass Errored Packet Enable.* When set, this bit indicates to the JT1001 controller that the HOST software wishes to receive errored packets. The errored packets are deposited (as best they can be) into HOST receive buffers and appropriate error bits are set in the receive header status fields.

Bit Field	Type	E^2	Mnemonic	Default Value	Description
19	RW	x	TXCREN	1	*Transmit CRC Enable.* When set, the JT1001 controller will generate and append a CRC to transmitted packets. When clear, the JT1001 controller does not generate or append the CRC to transmitted packets. Clearing the TXCREN bit can conflict and cause undefined behavior when other mode settings that cause the JT1001 controller to insert or modify packet data prior to the packets transmission. In particular, enabling the padding, VLAN tag insertion, or TCPIP checksum features when the TXCREN bit is clear, results in the packet being transmitted with a invalid CRC.
20	RW	√	DBMDEN	0	*Debug Mode Enable.* This bit enables (DBMDEN = 1) or disables debug mode.
21	RW	√	MGPKEN	0	*Magic Packet Enable.* This bit enables the JT1001 controller's ability to recognize a Magic Packet recognition and generate a wake up signal.
22	RW	√	MGMCBCEN	0	*Magic Packet Multicast/Broadcast Enable.* When clear, the JT1001 controller only accepts Magic Packet data sequences with a destination address that matches the LAN Physical Address CSR. When this bit is set, the JT1001 controller accepts Magic Packet data sequences whose destination address is a unicast, multicast address enabled via the *Multicast Hash Table Register,* or is an all stations broadcast.
23	RW	√	RXFLCTEN	1	*Receive Flow Control Enable.* This bit enables the JT1001 controller's ability to detect and act upon the reception of a MAC Control PAUSE frame. When this bit is set and the link is a full-duplex connection, the JT1001 controller will disable the transmitter when a MAC Control PAUSE frame is received. The JT1001 controller will re-enable the transmitter after the duration of time specified in the PAUSE frame has elapsed. If another PAUSE frame is received before time has elapsed, the JT1001 controller will reset its timer to the value specified in the subsequent PAUSE frame. When clear, the JT1001 controller will ignore PAUSE frames.
24	W	x	SERECL	0	*Set/Reset Control.* Set/reset control bit for bits[31:25].
25	RW	√	VLEN	0	*VLAN Enable.* When set, the JT1001 controller's VLAN support is enabled. When clear, VLAN support is disabled.
26	RW	√	VLTBEN	0	*VLAN Tag Table Enable.* This bit enables/disables the JT1001 controller's VLAN receive filter. The VLAN receive filter is applied to packets that contain a VLAN header and have already passed the destination address filter. When this bit is set, the JT1001 controller only accepts packets whose VLAN IDs are found in the VLAN register table. If VLTBEN is clear, the JT1001 controller accepts all VLAN frames that pass the destination address filter.

Bit Field	Type	E^2	Mnemonic	Default Value	Description
27	RW	√	VLRMID	0	*VLAN Remove ID.* When set, the JT1001 controller removes the VLAN header from received packets. When clear, the JT1001 controller does not remove the VLAN header from packets.
28	RW	√	VLISGB	0	*VLAN Insert Global.* When set, the JT1001 controller inserts the global VLAN header prior to transmitting a packet. The VLAN header is constructed based on the tag definition at index zero of the VLAN Tag Table. When clear, the JT1001 controller does not automatically insert a VLAN header into a transmitted packet. The effect of the VLISGB bit can be overridden on a per packet basis by setting the VLIS bit in the transmit header. Refer to the transmit header definitions for PIO, PDL, and PDC modes for more detail.
29	RW	√	USPIMD0	0	*User Pin0Mode.* This bit determines whether User Pin0 is an input or output. When this bit is set, User Pin0 is an input. When operating as an input, the JT1001 controller updates the USPIST0 bit in the *Chip Status Register* to reflect the state of this pin. When this bit is clear, the JT1001 controller drives the state of the pin based on the value in the USPIST0 bit in the *Chip Status Register.*
30	RW	√	USPIMD1	0	*User Pin1Mode.* This bit determines whether User Pin1 is an input or output. When this bit is set, User Pin1 is an input. When operating as an input, the JT1001 controller updates the USPIST1 bit in the *Chip Status Register* to reflect the state of this pin. When this bit is clear, the JT1001 controller drives the state of the pin based on the value in the USPIST1 bit in the *Chip Status Register.*
31	RW	√	LNCKEN	1	*Length Check Enable.* This bit governs the JT1001 controller's ability to detect length errors in received packets. A length error is defined as a packet containing a length/type field with a value less than 1536 and one of the following two conditions: • The value is greater than the number of bytes in the data field (the bytes after length/type and before the FCS). • The value is less than the number of bytes in the data field and the packet size is not the minimum length or greater than the maximum length (i.e., a large packet). When LNCKEN is set, the JT1001 controller's length checking logic is enabled. When LNCKEN is clear, the JT1001 controller's length checking logic is disabled.

CSR 01 MODE REGISTER – 2

31–29	28–24	23–21	20–19	18–16	15	14	13	12	11	10	9	8	7–6	5–4	3–2	1–0
RESRVD	RESRVD	PACKEREN	RXUPCKEN	RXTPCKEN	RXIPCKEN	TXUPCKEN	TXTPCKEN	TXIPCKEN	SERECL	LPBKMD	RESRVD					

Bit Field	Type	E²	Mnemonic	Default Value	Description
4:0	N/A	x	RESRVD	N/A	*Reserved.*
7:5	RW	x	LPBKMD	000	Loopback Mode. • 000 is no loopback. • 010 is MAC loopback. This setting causes the JT1001 controller's MAC block to route outbound packets to the MAC receiver logic instead of to the PHY transmitter logic. • 100 is PHY wireside loopback. This setting causes the JT1001 controller's PHY block to route inbound packets to the PHY transmitter logic instead of to the MAC receiver logic. All other values are reserved.
8	W	x	SERECL	0	*Set/Reset Control.* Set/reset control bit for bits[15:9].
9	RW	√	TXIPCKEN	0	*Transmit IP Header Checksum Enable.* When set, the JT1001 controller calculates and inserts the IP header checksum into all packets containing an IP header prior to the packet's transmission. When clear, the JT1001 controller does not calculate or insert the IP header checksum prior to the transmission of packets containing an IP header. The same effect can be accomplished on a per-packet basis by clearing the TXIPCKEN bit and setting the IPCKIS bit in the packet's transmit header. Refer to the transmit header definitions for PIO, PDL, and PDC modes for more detail.
10	RW	√	TXTPCKEN	0	*Transmit TCP Checksum Enable.* When set, the JT1001 controller calculates and inserts the TCP checksum into all packets containing a TCP header prior to the packet's transmission. When clear, the JT1001 controller does not calculate or insert the TCP checksum prior to the transmission of packets containing a TCP header. The same effect can be accomplished on a per-packet basis by clearing the TXTPCKEN bit and setting the TPCKIS bit in the packet's transmit header. Refer to the transmit header definitions for PIO, PDL, and PDC modes for more detail.

Bit Field	Type	E^2	Mnemonic	Default Value	Description
11	RW	√	TXUPCKEN	0	*Transmit UDP Checksum Enable.* When set, the JT1001 controller calculates and inserts the UDP checksum into all packets containing an UDP header prior to the packet's transmission. When clear, the JT1001 controller does not calculate or insert the UDP checksum prior to the transmission of packets containing an UDP header. The same effect can be accomplished on a per-packet basis by clearing the TXUPCKEN bit and setting the UPCKIS bit in the packet's transmit header. Refer to the transmit header definitions for PIO, PDL, and PDC modes for more detail.
12	RW	√	RXIPCKEN	0	*Receive IP Header Checksum Enable.* When set, the JT1001 controller computes the IP header checksum of all received packets containing an IP header and compares the checksum against the IP header checksum in the received packet. If the checksum passes, the packet is accepted and IPCKSMER bit in the receive header is cleared. If the checksum fails, the action taken depends on the state of the PACKEREN bit. If the PACKEREN bit is set, IPCKSMER bit is set in the receive header. Otherwise, the JT1001 controller discards the packet.
13	RW	√	RXTPCKEN	0	*Receive TCP Checksum Enable.* When set, the JT1001 controller computes the TCP checksum of all received packets containing a TCP header and compares the checksum against the TCP checksum in the received packet. If the checksum passes, the packet is accepted and TPCKSMER bit in the receive header is cleared. If the checksum fails, the action taken depends on the state of the PACKEREN bit. If the PACKEREN bit is set, TPCKSMER bit is set in the receive header. Otherwise, the JT1001 controller discards the packet.
14	RW	√	RXUPCKEN	0	*Receive UDP Checksum Enable.* When set, the JT1001 controller computes the UDP checksum of all received packets containing a UDP header and compares the checksum against the UDP checksum in the received packet. If the checksum passes, the packet is accepted and UPCKSMER bit in the receive header is cleared. If the checksum fails, the action taken depends on the state of the PACKEREN bit. If the PACKEREN bit is set, UPCKSMER bit is set in the receive header. Otherwise, the JT1001 controller discards the packet.

Bit Field	Type	E^2	Mnemonic	Default Value	Description
15	RW	√	PACKEREN	0	*Pass Checksum Error Enable.* This bit determines how the JT1001 controller handles packets containing IP, TCP, or UDP checksum errors. If this bit is set, the JT1001 controller sets the appropriate checksum error bits in the receive header and passes the packet to HOST software. If this bit is clear, the JT1001 controller discards the packet and HOST software never sees the packet. The PAERPKEN bit in *Mode Register – 1* overrides this bit. If PAERPKEN is set, packets containing TCP/IP checksum errors are received regardless of the setting of PACKEREN. If PAERPKEN is clear, the setting of PACKEREN governs the acceptance of packets TCP/IP checksum errors.
31:16	N/A	x	RESRVD	N/A	*Reserved.*

CSR 02 TRANSMIT PDC BUFFER ADDRESS TABLE INDEX

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RESRVD								RESRVD								RESRVD								RESRVD						TBIX			

Bit Field	Type	E^2	Mnemonic	Default Value	Description
5:0	W	x	TBIX	0	*Table Index.* This value specifies which slot in the PDC Transmit Base Address Table will be modified by the next write to the *Transmit PDC Buffer Address Register.* The PDC Transmit Base Address Table has 64 slots.
7	x	x	RESRVD	x	*Reserved.*
31:8	x	x	RESRVD	x	*Reserved.*

In PDC mode, a table of pre-allocated buffer physical addresses is maintained onboard the JT1001 controller. PDC transmit commands passed to the JT1001 controller reference preprogrammed addresses in the table with the buffer ID (BID) field of the PDC command. The BID field is used to index into the table and extract the address of the desired buffer. Thus, the JT1001 controller knows where in HOST memory to look for the data to be transmitted.

As used above, pre-allocated buffers are locked (non-pageable), and occupy contiguous CPU pages (allocated once per driver invocation) and their physical addresses can be determined far in advance of actual use — usually at system initialization time. For this process to work correctly, the physical addresses of the pre-allocated buffers must be made known to the JT1001 controller prior to first use (typically during initialization). HOST software initializes the table by writing the index of the desired table entry into this register and the physical address of a specific pre-allocated buffer into the *Transmit PDC Buffer Address*

MSD Register and the *Transmit PDC Buffer Address LSD Register*. The stated order is required for addresses larger than 32-bits on PCI implementations that do not support 64-bit I/O — essentially, all current PCI implementations. If PCI versions subsequent to 2.1 implement 64-bit I/O, the *Transmit PDC Buffer Address LSD/MSD Register* can be accessed as a single register and the sequence restriction does not apply. A method for reducing the number of I/O cycles required to pass addresses larger than 32-bits using a single PCI transaction in systems supporting only 32-bit I/O is described below. By repeating this process for each slot in the table, the whole table can be initialized with the physical addresses of pre-allocated buffers. The JT1001 controller automatically increments TBIX after each write to the *Transmit PDC Buffer Address LSD Register.* If HOST software is initializing the table slots sequentially, it need only write the initial TBIX value.

For systems that do not require the additional 32-bits of address space, or for those systems where the upper 32-bits are the same for all the transmit buffers, the MSD register can be written once with the appropriate value. Once the MSD register is initialized, all subsequent writes can be directed at the LSD register. Each time the LSD register is written, the JT1001 controller will transfer the preprogrammed value in the MSD register along with the new value programmed into the LSD register into the buffer address table.

CSR 03 PRODUCT IDENTIFICATION REGISTER

31–24	23–16	15–0
RESRVD	RVID	DVID

Bit Field	Type	E²	Mnemonic	Default Value	Description
15:0	R	√	DVID	1	*Device Identifier.* A value that uniquely identifies the JT1001 controller from all other Jato Technologies products. When read, this field returns the same value that is returned when reading the Device ID register in the JT1001 controller's PCI configuration space.
23:16	R	x	RVID	0	*Revision Identifier.* A value that uniquely identifies a particular revision level of the JT1001 controller. Revision numbers are assigned beginning at 0. When read, this field returns the same value that is returned when reading the *Revision ID Register* in the JT1001 controller's PCI configuration space.
31:24	x	x	RESRVD	0	*Reserved.*

CSR 04 TRANSMIT PDC BUFFER ADDRESS LSD

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BFADLO																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	W	x	BFADLO	0	*Buffer Address Low DWORD.* Values written to this register are placed by the JT1001 controller into the LSD (bits 31:0) of the Transmit PDC Buffer Address Table. Once HOST software writes to this register, the JT1001 controller will transfer the contents of this register and the contents of the register to the Transmit PDC Buffer Address Table. The HOST identifies which entry in the buffer address table is to be modified by writing the entry's index value in the *Transmit PDC Buffer Address Table Index Register.*

CSR 05 TRANSMIT PDC BUFFER ADDRESS MSD

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BFADHI																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	W	x	BFADHI	0	*Buffer Address High DWORD.* Values written to this register are placed by the JT1001 controller into the MSD (bits 63:32) of the Transmit PDC Buffer Address Table. The HOST identifies which entry in the buffer address table is to be modified by writing the entry's index value in the *Transmit PDC Buffer Address Table Index Register.* The buffer address table proper is not affected by any data stored in this register until the LSD (bits 31:0) of the address are written into the *Transmit PDC Buffer Address LSD Register.*

CSR 06 RECEIVE PDC BUFFER ADDRESS TABLE INDEX

31–08	07	06–00
RESRVD	RESRVD	TBIX

Bit Field	Type	E^2	Mnemonic	Default Value	Description
5:0	W	x	TBIX	0	*Table Index*. This value specifies which slot in the PDC Receive Base Address Table will be modified by the next write to the *PDC Receive Base Address Table Data Register*. The PDC Receive Base Address Table has 64 slots.
7	x	x	RESRVD	x	*Reserved.*
31:8	x	x	RESRVD	x	*Reserved.*

In PDC mode, a table of pre-allocated buffer physical addresses is maintained onboard the JT1001 controller. PDC receive commands passed to the JT1001 controller reference pre-programmed addresses in the table with the BID field of the PDC command. Thus, the JT1001 controller knows where in HOST memory to look for the data to be transferred.

As used above, pre-allocated buffers are locked (non-pageable), and occupy contiguous CPU pages (allocated once per driver invocation) and their physical addresses can be determined far in advance of actual use — usually at system initialization time. For this process to work correctly, the physical addresses of the pre-allocated buffers must be made known to the JT1001 controller prior to first use (typically during initialization). HOST software initializes the table by writing the index of the desired table entry into this register and the physical address of a specific pre-allocated buffer into the *Receive PDC Buffer Address MSD Register* and the *Receive PDC Buffer Address LSD Register*. The stated order is required for addresses larger than 32 bits on PCI implementations that do not support 64-bit I/O — essentially, all current PCI implementations. If PCI versions subsequent to 2.1 implement 64-bit I/O, the *Receive PDC Buffer Address LSD/MSD Register* can be accessed as a single register and the sequence restriction does not apply. A method for reducing the number of I/O cycles required to pass addresses larger than 32-bits using a single PCI transaction in systems supporting only 32-bit I/O is described below. By repeating this process for each slot in the table, the whole table can be initialized with the physical addresses of pre-allocated buffers. The JT1001 controller automatically increments TBIX after each write to the *Receive PDC Buffer Address LSD Register*. If HOST software is initializing the table slots sequentially, it need only write the initial TBIX value.

For systems that do not require the additional 32 bits of address space, or for those systems where the upper 32 bits are the same for all the transmit buffers, the MSD register can be written once with the appropriate value. Once the MSD register is initialized, all subsequent writes can be directed at the LSD register. Each time the LSD register is written, the JT1001 controller will transfer the

preprogrammed value in the MSD register along with the new value programmed into the LSD register into the buffer address table.

CSR 07 RESERVED

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RESRVD																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	x	x	RESRVD	0	*Reserved.*

CSR08 RECEIVE PDC BUFFER ADDRESS LSD

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
BFADLO																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	W	x	BFADLO	0	*Buffer Address Low DWORD.* Values written to this register are placed by the JT1001 controller into the Receive PDC Buffer Address Table at the slot indicated by the TBIX field in the *Receive PDC Buffer Address Table Register.* The JT1001 controller examines the values in the *Receive PDC Buffer Address LSD/MSD Registers* when the *Receive PDC Buffer Address LSD Register* is written.

CSR 09 RECEIVE PDC BUFFER ADDRESS MSD

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
BFADHI																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	W	x	BFADHIDW	0	*Buffer Address High DWORD.* Values written to this register are placed by the JT1001 controller into the Receive PDC Buffer Address Table at the slot indicated by the TBIX field in the *Receive PDC Buffer Address Table Register.* The JT1001 controller examines the values in the *Receive PDC Buffer Address LSD/MSD Registers* when the *Receive PDC Buffer Address LSD Register* is written. The buffer address table proper is not affected by any data stored in this register until the LSD (bits 31:0) of the address is written into the *Receive PDC Buffer Address LSD Register.*

CSR 10 EEPROM REGISTER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RESRVD								RESRVD		FLWTEN	FLPN	EXRMTM				RESRVD		EEAD						RESRVD	EESL	EEMU	EECKSME	RESRVD	EEWTCM	EERDCM	EEPMPN

Bit Field	Type	E^2	Mnemonic	Default Value	Description
0	R	x	EEPMPN	0	*EEPROM Present.* This bit is set if EEPROM is present.
1	W	x	EERDCM	1	*EEPROM Read Command.* This bit selects the EEPROM read command. This bit must be set in conjunction with either the EEMU bit or EESL bit for a read command to occur.
2	W	x	EEWTCM	0	*EEPROM Write Command.* This bit selects the EEPROM write command. This bit must be set in conjunction with the EESL bit for a write command to occur.
3	x	x	RESRVD	0	*Reserved.*
4	A	x	EECKSMER	0	*EEPROM Checksum Error.* When set, this bit indicates that a problem occurred when the JT1001 attempted to read EEPROM. HOST software can retry the operation if desired.

Bit Field	Type	E^2	Mnemonic	Default Value	Description
5	WA	x	EEMU	1	*EEPROM Multiple Access.* Instructs the JT1001 controller to begin processing a multiple access read command. A multiple access read command causes the JT1001 controller to reinitialize CSRs with their associated EEPROM values. See Figure 7-1 for a list of CSRs that have associated EEPROM values. The EEMU bit must be set in conjunction with the EERDCM bit for the reads to occur. The EEMU bit remains set for the duration of the multiple access read command. Upon completion of the command, the JT1001 controller will automatically clear this bit. HOST software can poll this bit to determine when the command has completed. Setting the EEMU bit in conjunction with the EEWTCM bit will result in an error. More specifically, no action will be taken by the JT1001 controller except to set the EECMER bit. The default value for this register is 1. This causes the JT1001 controller to read EEPROM following a hard or soft reset. When the read of EEPROM has completed, the JT1001 controller clears EEMU.
6	WA	x	EESL	0	*EEPROM Single Access.* Instructs the JT1001 controller to begin processing a single access command to EEPROM. A single access command can be either a read or write. Setting the EESL bit in conjunction with the EERDCM bit causes a single access read from EEPROM. The EEPROM offset read from EEPROM is specified by EEAD. The JT1001 controller places the value read from EEPROM into the *EEPROM Data Register*. Setting the EESL bit in conjunction with the EEWTCM bit causes a single access write to EEPROM. The EEPROM offset written is specified by EEAD. The *EEPROM Data Register* contains the value to be written. Whether reading or writing, the EESL bit remains set for the duration of the command. Upon completion of the command, the JT1001 controller automatically clears this bit. HOST software can poll this bit to determine when the command has completed.
7	x	x	RESRVD	0	*Reserved.*
12:8	W	x	EEAD	0	*EEPROM Address.* This field specifies the offset of the EEPROM DWORD to be read or written. Although EEPROM is organized in 16-bit WORDs, the JT1001 controller presents EEPROM to software as 32-bit DWORDs. For example, to read the second DWORD in EEPROM, software will set EEAD to 2. The JT1001 controller will then retrieve the WORDs at EEPROM offsets 4 and 5 and place the result in the *EEPROM Data Register*.
15:13	x	x	RESRVD	0	*Reserved.*
19:16	RW	√	EXRMTM	0	*Expansion ROM Timings.*

Bit Field	Type	E^2	Mnemonic	Default Value	Description
20	R	√	FLPN	0	*Flash Present.* This bit is used to distinguish between the presence of a flash device or ROM device. When set, this bit indicates a flash device is attached to the JT1001 controller. When clear, a ROM device is attached.
21	RW	√	FLWTEN	0	*Flash Write Enable.* When set, the JT1001 controller allows the flash expansion ROM to be written to via PCI memory transactions. When clear, the flash can not be written.
31:22	x	x	RESRVD	0	*Reserved.*

The EEPROM is manipulated at the following times:

1. During manufacturing, when the default configuration and the MAC address are programmed for the first time.
2. Each time the system is restarted, the default configuration is reloaded into volatile JT1001 controller registers.
3. Anytime HOST software explicitly requests that EEPROM be read or written. This might happen if a utility program is used to change the programmed significance of the LEDs.

EEPROM is used as a convenient nonvolatile store of JT1001 controller parameters that are directly related to a particular JT1001 controller and ought not easily change, or that are required during system boot. Apart from system startup time, EEPROM will seldom be manipulated. EEPROM may also be used by HOST software as a nonvolatile store for software configuration parameters.

When reloading CSRs from EEPROM, the JT1001 controller calculates a 32-bit checksum and compares the checksum against the checksum value stored in EEPROM. If HOST software writes a value in EEPROM, it must also recalculate and write the new checksum value to EEPROM.

CSR 11 CHIP STATUS REGISTER

31–25	24–16	15–8	7	6	5	4	3	2	1	0
RESRVD	RESRVD	RESRVD	RESRVD	RXIL	TXIL	USPIST1	USPIST0	FLCTST	RXPKAV	TXPKAC

Bit Field	Type	E^2	Mnemonic	Default Value	Description
0	R	x	TXPKAC	1	*Transmit Packet Acceptable.* When set, indicates that the TX FIFO has room for at least one maximum size packet.
1	R	x	RXPKAV	0	*Receive Packet Available.* When set, indicates that the RX FIFO holds at least one packet.
2	RA	x	FLCTST	0	*Flow Control Status.* The JT1001 controller sets this bit when it has temporarily disabled the transmitter due to the reception of a PAUSE frame. The JT1001 controller clears this bit when it has re-enabled the transmitter. HOST software queries this bit to determine if the transmitter has been disabled due to reception of a PAUSE frame.
3	RW	x	USPIST0	x	*User Pin0 State.* The meaning and purpose of this bit varies depending on the state of the USPMD0 bit in *Mode Register – 1*. When USPMD0 is set, User Pin0 is an input pin. In this case, HOST software can read USPIST0 to determine the state of User Pin0. If User Pin0 is high, the JT1001 controller sets USPIST0. If User Pin0 is low, the JT1001 controller clears USPIST0. HOST software must not write to USPIST0 when User Pin0 is acting as an input pin. When USPMD0 is clear, User Pin0 is an output pin. In this case, the JT1001 controller drives the state of User Pin0 according to state of USPIST0. If HOST software sets USPIST0 to 1, the JT1001 controller drives User Pin0 to the high state. If HOST software clears USPIST0, the JT1001 controller drives User Pin0 to the low state.

Bit Field	Type	E^2	Mnemonic	Default Value	Description
4	RW	x	USPIST1	x	*User Pin1 State.* The meaning and purpose of this bit varies depending on the state of the USPMD1 bit in *Mode Register – 1*. When USPMD1 is set, User Pin1 is an input pin. In this case, HOST software can read USPIST1 to determine the state of User Pin1. If User Pin1 is high, the JT1001 controller sets USPIST1. If User Pin1 is low, the JT1001 controller clears USPIST1. HOST software must not write to USPIST1 when User Pin1 is acting as an input pin. When USPMD1 is clear, User Pin1 is an output pin. In this case, the JT1001 controller drives the state of User Pin1 according to state of USPIST1. If HOST software sets USPIST1 to 1, the JT1001 controller drives User Pin1 to the high state. If HOST software clears USPIST1, the JT1001 controller drives User Pin1 to the low state.
5	R	x	TXIL	1	*Transmitter Idle.* This bit indicates whether or not the JT1001 controller's transmitter is currently transmitting a packet. When set, the JT1001 controller is not currently transmitting a packet. When clear, the JT1001 controller is currently transmitting a packet.
6	R	x	RXIL	1	*Receiver Idle.* This bit indicates whether or not the JT1001 controller's receiver is currently receiving a packet. When set, the JT1001 controller is not currently receiving a packet. When clear, the JT1001 controller is currently receiving a packet.
31:7	x	x	RESRVD	x	*Reserved.*

CSR12 TRANSMIT PDL ADDRESS REGISTER LSD

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PKDSADLO																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	W	x	PKDSADLO	N/A	*Descriptor Address.* Writing to this register loads a PDL's address into the JT1001 controller's TX command FIFO and increments the TX command FIFO count. These actions are triggered when the least significant DWORD is written.

Figure 5-1 describes the format of the PDL used in the master mode packet transmission. The descriptor is used to describe a single packet. HOST software guarantees the PDLs are QWORD aligned. The fragments pointed to by PDLs can be on any byte boundary.

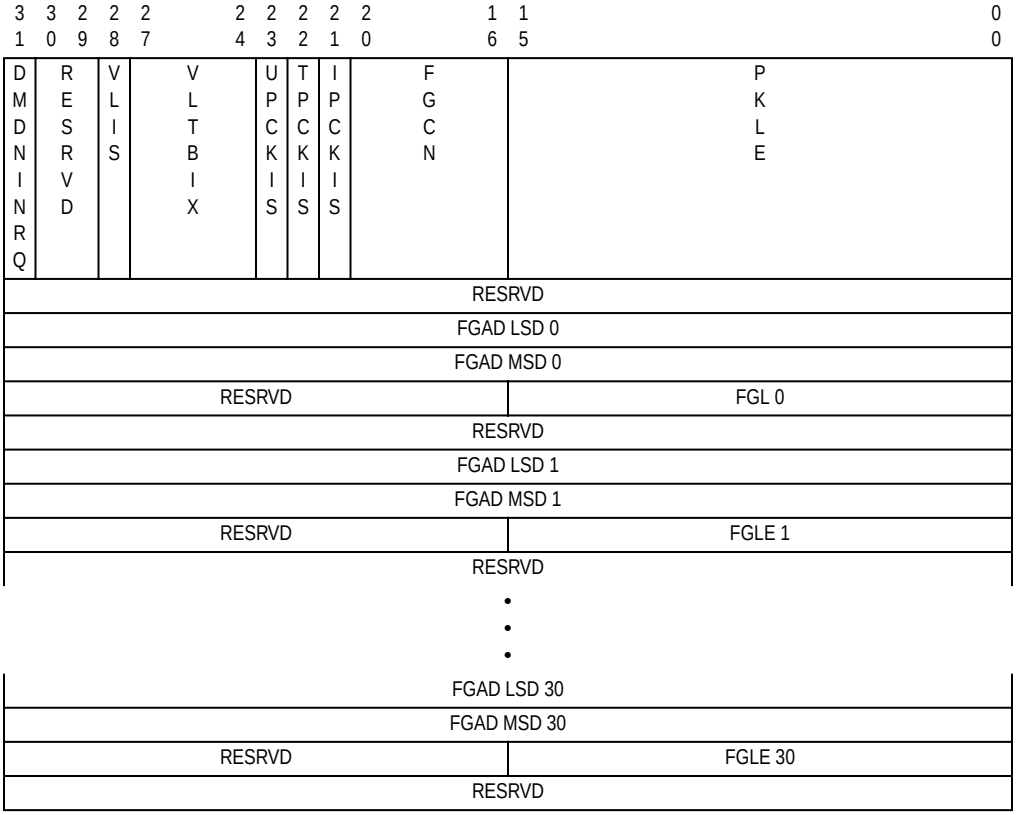


Figure 5-1. PDL Transmit Header Format

The fields of the packet descriptor have the following significance:

PKLE — *Packet Length.* The length of the packet contained in the fragments described by this data structure is reflected in the packet length field. As the BMC processes this data structure, it takes the value in the PKLE field and passes it to the MAC to indicate the number of bytes to be transmitted. PKLE must equal the sum of the FGLEx fields. If it does not, the packet is not transmitted onto the wire. The maximum value for PKLE is 32 Kbytes — the size of the TX FIFO packet header. HOST software must guarantee that this maximum is not exceeded.

FGCN — *Fragment Count.* Indicates the number of fragments that are defined by the PDL. The sum of the lengths of each fragment in the PDL corresponds with the value stored in the PKLE field. This field allows the BMC to determine the size of the PDL. A maximum size PDL can accommodate up to 31 fragments. Each fragment descriptor is 16 bytes in length. Thus, the maximum length PDL can be 16 * 31 + 8 = 504 bytes in length.

IPCKIS — *IP Header Checksum Insert.* This bit allows HOST software to request the IP header checksum be inserted in the packet. Setting this bit causes the

JT1001 controller to calculate and insert the IP header checksum into the packet that contains an IP header. If the packet does not contain an IP header, the JT1001 controller does not calculate or insert the checksum.

TPCKIS — *TCP Checksum Insert.* This bit allows HOST software to request the TCP checksum be inserted into the packet. Setting this bit causes the JT1001 controller to calculate and insert the TCP checksum into a packet that contains a TCP header. If the packet does not contain a TCP header, the JT1001 controller does not calculate or insert the checksum.

UPCKIS — *UDP Checksum Insert*. This bit allows HOST software to request the UDP checksum be inserted into the packet. Setting this bit causes the JT1001 controller to calculate and insert the UDP checksum into the packet that contains a UDP header. If the packet does not contain a UDP header, the JT1001 controller does not calculate or insert the checksum.

VLTBIX — *VLAN TCI Table Index.* This field is an index into the VLAN TCI Table. The JT1001 controller uses the TCI information at this index to construct a VLAN tag header.

VLIS — *VLAN Insert Tag*. Setting this bit causes the JT1001 controller to construct and insert a VLAN tag header into the packet prior to its transmission. The JT1001 controller constructs the VLAN tag header using the TCI at index VLTBIX in the VLAN TCI Table.

DMDNINRQ — *DMA Done Interrupt Request.* Indicates to the BMC that an interrupt is requested when the BMC is done transferring the final fragment described by the PDL. Note that no explicit correlation exists between the transferred data and the interrupt.

FGAD — *Fragment Address.* Fields used to pass the physical addresses of fragments to the BMC. Each PDL can accommodate up to 31 fragment addresses arranged in physically contiguous memory immediately following the PDL's command field.

FGLE — *Fragment Length.* Fields used to denote the length in bytes of the packet data stored in the fragment. The sum of all the *FGLE* fields in a PDL equals the value specified in the *PKLE* field.

RESRVD — *Reserved.* HOST software sets this field to 0.

CSR 13 TRANSMIT PDL ADDRESS REGISTER MSD

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
PKDSADHI																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	W	x	PKDSADHI	N/A	*Packet Descriptor Address High.* Writing to this register loads the address of a PDL into the JT1001 controller's Transmit Command FIFO and increments the Transmit Command FIFO Count. These actions are triggered when the least significant. DWORD is written. Writing to this register does not initiate a Transmit PDL command. Transmit PDL commands are only initiated when the LSD of the PDL Address is written into the *Transmit PDL Address LSD Register.*

CSR 14 RECEIVE PDL ADDRESS REGISTER LSD

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
PKDSADLO																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	W	x	PKDSADLO	N/A	*Packet Descriptor Address Low.* Writing to this register loads the address of a PDL into the JT1001 controller's Receive Command FIFO and increments the Receive Command FIFO Count. These actions are triggered when the least significant DWORD is written.

The *Receive PDL Address Register* accepts pointers to receive PDLs. These PDLs have a reciprocal function to their transmit counterparts. However, the format of the command block is nearly identical.

Figures 5-2 and 5-3 denote the positions and names of the bits in the receive PDL header. Note that for receive PDLs, there are two distinct packet formats. The first is used when the PDL is transferred to the JT1001 controller. This pre-receive header format is used to inform the JT1001 controller of the location of receive buffers and the *per PDL* processing options that the HOST software

wishes to enable. When the JT1001 controller transfers a packet into HOST memory, the post-receive PDL header format is used to convey the packet length and reception status to the HOST. HOST software guarantees the PDLs are QWORD aligned. The fragments pointed to by PDLs can be on any byte boundary.

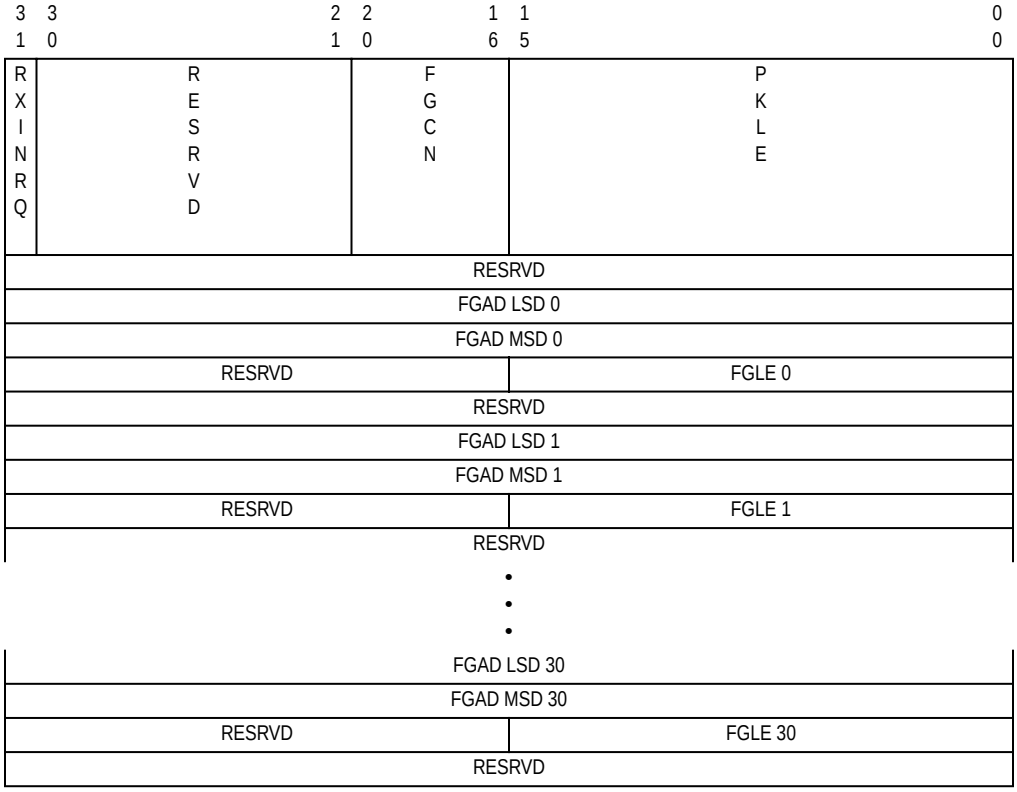


Figure 5-2. PDL Pre-Receive Header Format

Pre-receive PDL header fields are initialized by HOST software prior to handing the PDL to the JT1001 controller. The fields of the pre-receive packet descriptor have the following definition:

PKLE — *Packet Length.* The maximum number of received data bytes the packet descriptor can accommodate. This value is the sum of the individual fragment lengths. If PKLE is greater than the sum of the fragments and the received packet size is greater than the sum of the fragment lengths, the received packet is truncated and no error is indicated. If PKLE is less than the sum of the fragments and the packet size is greater than PKLE, the EROV in the post-receive PDL is set to indicate an overflow error. The maximum value for PKLE is 64 Kbytes — the size of the RX FIFO packet header.

FGCN — *Fragment Count.* The number of fragments attached to the PDL. The maximum number of fragments is 31. Each fragment descriptor is 16 bytes in length. Thus, the maximum length PDL can be 16 * 31 + 8 = 504 bytes in length.

RXINRQ — *Receive Interrupt Request.* When set, this bit forces a receive interrupt to be generated when the JT1001 controller finishes transferring the packet to HOST memory, even if the RXMS bit in the *Interrupt Mask Register*

is reset. The interrupt is a one-time interrupt associated with the PDL that has the RXINRQ bit set. This bit is useful in reducing the overall number of interrupts passed to the HOST for receive packet processing.

FGAD — *Fragment Address.* Fields used to pass the physical addresses of fragments to the BMC. Each PDL can accommodate up to 31 fragment addresses arranged in physically contiguous memory immediately following the PDL's command field.

FGLE — *Fragment Length.* Fields used to specify the maximum number of bytes each fragment can accommodate. The sum of all the FGLE fields in a PDL equals the value specified in the PKLE field.

RESRVD — *Reserved.* HOST software sets this field to 0.

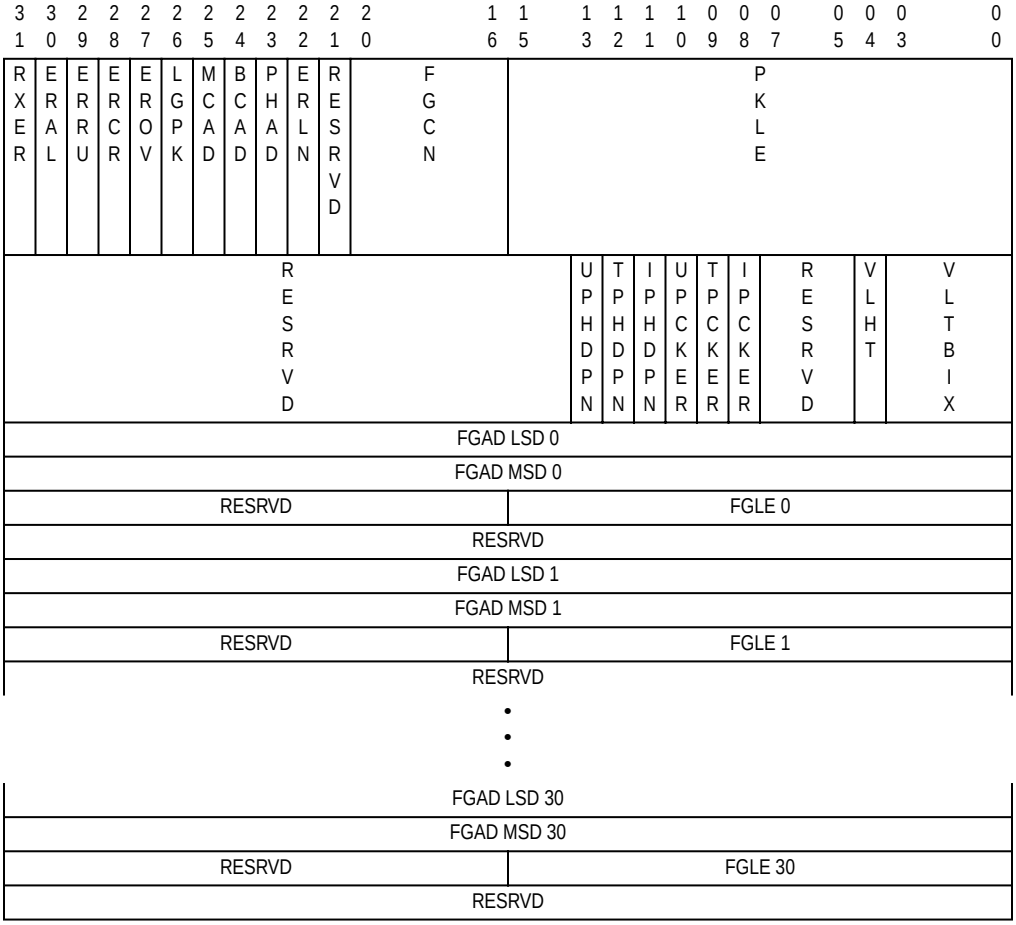


Figure 5-3. PDL Post-Receive Header Format

Once the JT1001 controller has deposited a received packet into the memory described by a PDL, the JT1001 controller updates the PDL header to contain the packet length and receive status. The post-receive PDL header fields are defined as follows:

PKLE — *Packet Length.* Indicates the packet's length in bytes. Note that this value reflects the actual length of the packet as determined by the JT1001

controller. In cases where the packet overflows the receive buffer, this field still reflects the length of the packet and not the amount of data deposited into the buffer.

FGCN — *Fragment Count.* The number of fragments attached to the PDL. The value of this field remains unchanged from the pre-receive PDL header.

PHAD — *Physical Address.* The PHAD bit indicates that the received packet's destination address matches the JT1001 controller's station (MAC) address.

BCAD — *Broadcast Address.* This bit indicates that the received packet's destination address was the broadcast address.

MCAD — *Multicast Address.* When the JT1001 controller multicast address filtering mechanism determines that a packet with a multicast destination address should be passed to the HOST, it sets this bit in the PDL header and transfers the packet to HOST memory.

LGPK — *Large Packet.* By setting this bit, the JT1001 controller indicates that the inbound packet was determined to be larger than the maximum allowable length for an ethernet frame. If the LGPKEN enable bit is clear, the JT1001 controller regards this condition as an error. If the LGPKEN enable bit is set, the JT1001 controller does not regard this condition as an error.

EROV — *Overflow Error.* It is possible for incoming data to exceed the space allotted to receive it. In such cases, the JT1001 controller will deliver as much data as will fit into the available buffer space. Any data that does not fit will be discarded. When this situation occurs, the JT1001 controller sets the EROV bit.

ERCR — *CRC Error.* When the JT1001 controller detects that an inbound packet's CRC does not match the computed value, it sets this bit to signal the condition.

ERRU — *Runt Error.* If the JT1001 controller determines that an inbound packet is shorter than the minimum ethernet packet length, it sets the ERRU bit.

ERAL — *Alignment Error.* This bit is set when the JT1001 controller receives a packet that is not an integral number of octets in length.

ERLN — *Length Error.* This bit is set when the JT1001 controller detects that an inbound packet's LLC data is shorter than the length specified in the length/type field of the packet's MAC header.

RXER — *Receive Error.* Whenever an error condition is detected for a received packet corresponding to a particular PDL, the error bit in that PDL is set to 1. This error bit is the "OR" of the ERLN, EROV, ERCR, ERRU, and ERAL bits. This bit is also set if the LGPKEN bit in *Mode Register – 1* is clear, the PAERPKEN bit in *Mode Register – 1* is set, and the LGPK bit is set.

VLTBIX — *VLAN Table Index.* This field indicates the index of the VLAN TCI Table entry that matched the TCI in the received packet's VLAN tag header. This field only has meaning if the VLHT bit is set.

VLHT — *VLAN Hit.* When set, this bit indicates the received packet contained a VLAN tag header whose TCI matched an entry in the VLAN TCI Table. This bit is set by the JT1001 controller if the VLEN and VLTBEN bits in *Mode Register – 1* are set and the VLAN tag information in the packet matches an entry in the VLAN TCI Table; otherwise this bit will not be set.

IPCKER — *IP Header Checksum Error.* When set, this bit indicates the packet failed the IP header checksum test. The JT1001 controller tests the IP header checksum in packets when the RXIPCKEN bit is set in *Mode Register – 2* and the packet contains an IP header. When clear, the packet either passed the IP header checksum test, did not contain an IP header, or the JT1001 controller's checksum support is disabled.

TPCKER — *TCP Checksum Error.* When set, this bit indicates the packet failed the TCP checksum test. The JT1001 controller tests the TCP checksum in packets when the RXTPCKEN bit is set in *Mode Register – 2* and the packet contains a TCP header and data. When clear, the packet either passed the TCP checksum test, did not contain a TCP header, or the JT1001 controller's checksum support is disabled.

UPCKER — *UDP Checksum Error.* When set, this bit indicates the packet failed the UDP checksum test. The JT1001 controller tests the UDP checksum in packets when the RXUPCKEN bit is set in *Mode Register – 2* and the packet contains a UDP header and data. When clear, the packet either passed the UDP checksum test, did not contain an UDP header, or the JT1001 controller's checksum support is disabled.

IPHDPN — *IP Header Present.* When set, this bit indicates the JT1001 controller found an IP header in the packet. When clear, the packet did not contain an IP header. The value of this bit is valid irrespective of the setting of the RXIPCKEN bit in *Mode Register – 2*.

TPHDPN — *TCP Header Present.* When set, this bit indicates the JT1001 controller found a TCP header in the packet. When clear, the packet did not contain a TCP header. The value of this bit is valid irrespective of the setting of the RXTPCKEN bit in *Mode Register – 2*.

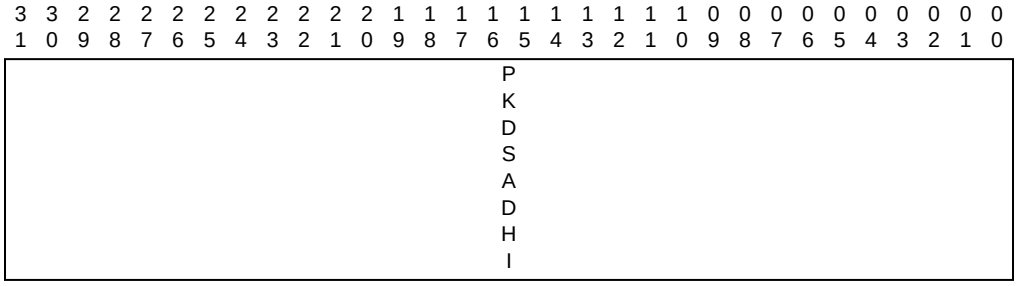
UPHDPN — *UDP Header Present.* When set, this bit indicates the JT1001 controller found an UDP header in the packet. When clear, the packet did not contain an UDP header. The value of this bit is valid irrespective of the setting of the RXUPCKEN bit in *Mode Register – 2*.

FGAD — *Fragment Address.* Fields used to pass the physical addresses of fragments to the BMC. The values of the FGAD fields remain unchanged from the pre-receive PDL header.

FGLE — *Fragment Length.* Fields used to specify the maximum number of bytes each fragment can accommodate. The values of the FGLE fields remain unchanged from the pre-receive PDL header.

RESRVD — *Reserved.* The JT1001 controller sets this field to 0.

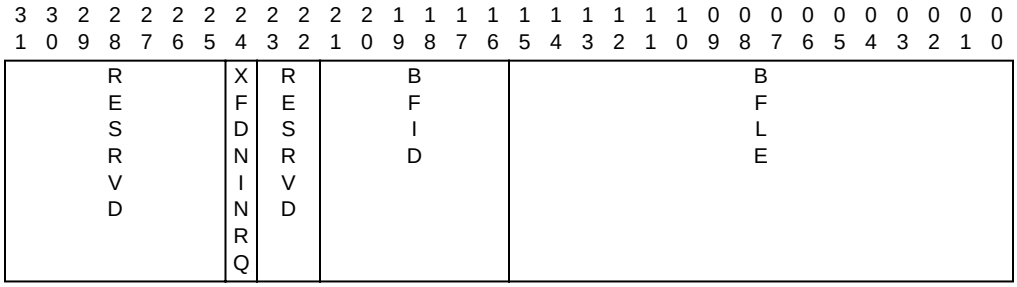
CSR 15 RECEIVE PDL ADDRESS REGISTER MSD



Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	W	x	PKDSADHI	N/A	*Packet Descriptor Address High.* Writing to this register loads bits 63:32 of a 64-bit PDL address into the JT1001 controller's *RX Command FIFO Staging Register*. When the LSD of the address is written (bits 31:0), the complete address is moved to the receive command FIFO and the *RX Command FIFO Count Register* is incremented. Writing to this register does not initiate a receive PDL command. Receive PDL commands are only initiated when the LSD of the PDL address is written into the *Receive PDL Address LSD Register.*

CSR16 TRANSMIT PDC REGISTER

Writing to this register initiates a packet Propulsion transfer for packet transmission.



Bit Field	Type	E^2	Mnemonic	Default Value	Description
15:0	W	x	BFLE	0	*Buffer Length.* Number of bytes to be transferred by the JT1001 controller.
21:16	W	x	BFID	0	*Buffer ID.* Uniquely identifies which pre-allocated HOST buffer to use for transmission.
23:22	x	x	RESRVD	0	*Reserved.*
24	W	x	XFDNINRQ	0	*Transfer Done Interrupt Request.* When set, this bit indicates that the HOST software wishes an interrupt to be generated upon completion of data transfer from HOST memory to JT1001 controller memory.
31:25	x	x	RESRVD	0	*Reserved.*

A transmit PDC buffer is used for PDC mode packet transmission. HOST software fills the transmit PDC buffer with one or more transmit requests and then enqueues the PDC buffer to the JT1001 controller via the *Transmit PDC Register*. HOST software guarantees the transmit PDC buffers are QWORD aligned. The maximum size of a transmit PDC buffer is 32 Kbytes. HOST software must guarantee that the maximum size is not exceeded.

Each transmission request within a transmit PDC buffer is identified by a transmit packet header. The initial DWORD in each transmit PDC buffer is a transmit packet header. The transmit data immediately follows the transmit packet header. The JT1001 controller can determine the type of a header by examining the HDTYPE field. The JT1001 controller can use the LEN field to determine the offset of the next header in the buffer. All headers within a transmit PDC buffer are aligned on a QWORD boundary. Currently, the only header type defined for transmit PDC buffers is the transmit packet header.

Each transmission request within a transmit PDC buffer consists of a transmit packet header followed by the data bytes that constitute the packet to be transmitted. The JT1001 controller can use the LEN field to determine the offset of the next header in the buffer. All transmit packet headers within a transmit PDC buffer begin on a QWORD boundary. A detailed description of the PDC transmit header and data format follows (see Figure 5-4).

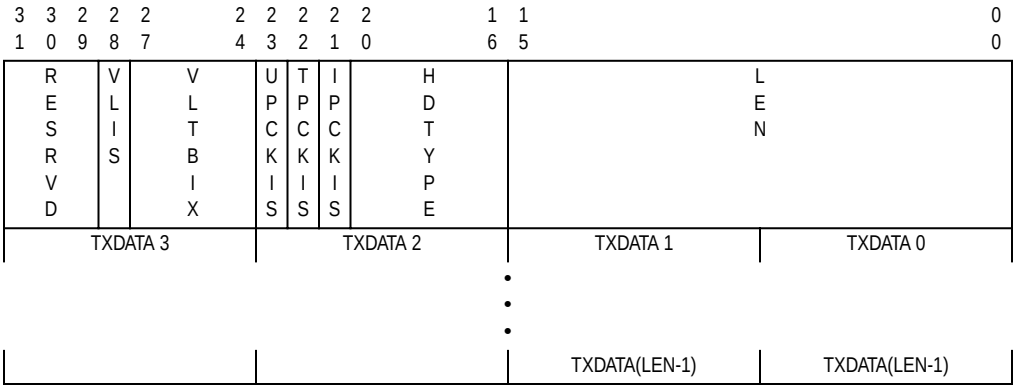


Figure 5-4. PDC Transmit Header and Data Format

The definition of the PDC transmit header and data fields follow.

LEN — *Length.* Indicates the number of bytes to be transmitted. LEN bytes of transmit data follow the transmit packet header.

HDTYPE — *Header Type.* A unique value used to identify the header type. For a transmit request header, the value is 0x2.

IPCKIS — *IP Header Checksum Insert.* This bit allows HOST software to request the IP header checksum be inserted into the packet. Setting this bit causes the JT1001 controller to calculate and insert the IP header checksum into the packet that contains an IP header. If the packet does not contain an IP header, the JT1001 controller does not calculate or insert the checksum.

TPCKIS — *TCP Checksum Insert*. This bit allows HOST software to request the TCP checksum be inserted in the packet. Setting this bit causes the JT1001 controller to calculate and insert the TCP checksum into a packet that contains a TCP header. If the packet does not contain a TCP header, the JT1001 controller does not calculate or insert the checksum.

UPCKIS — *UDP Checksum Insert.* This bit allows HOST software to request the UDP checksum be inserted into the packet. Setting this bit causes the JT1001 controller to calculate and insert the UDP checksum into the packet that contains a UDP header. If the packet does not contain a UDP header, the JT1001 controller does not calculate or insert the checksum.

VLTBIX — *VLAN TCI Table Index*. This field is an index into the VLAN TCI Table. The JT1001 controller uses the TCI information at this index to construct a VLAN tag header

VLIS — *VLAN Insert Tag Header*. Setting this bit causes the JT1001 controller to construct and insert a VLAN tag header into the packet prior to its transmission. The JT1001 controller constructs the VLAN tag header using the TCI at index VLTBIX in the VLAN TCI Table.

TXDATA — *Transmit Data.* Data bytes that constitute the packet to be transmitted. TXDATA is padded by HOST software to the next DWORD boundary.

CSR 17 RECEIVE PDC REGISTER

Writing to this register allows a packet Propulsion transfer for reception to occur at a future time.

31	30 29 28 27 26 25 24 23 22 21 20	19 18 17 16 15 14 13 12 11 10	09 08 07 06 05 04 03 02 01 00
RXINRQ	RESRVD	BFID	BFLE

Bit Field	Type	E^2	Mnemonic	Default Value	Description
15:0	W	x	BFLE	0	*Buffer Length*. The maximum number of bytes that can be transferred into the buffer specified at BFID.
21:16	W	x	BFID	0	*Buffer ID*. Uniquely identifies which pre-allocated HOST buffer to use for reception.
30:22	W	x	RESRVD	0	*Reserved*.
31	W	x	RXINRQ	0	*Receive Interrupt Request.* This bit communicates to the JT1001 controller that the HOST wishes to be interrupted when a packet is received into the buffer corresponding to this PDC command. The interrupt will be generated regardless of the state of the RXMS bit in the *Interrupt Mask Register.*

A receive PDC buffer is used for PDC mode packet reception. The *Receive PDC Register* is used to enqueue a receive PDC buffer to the JT1001 controller. The JT1001 controller fills the receive PDC buffer with data representing one or more received packets. When enqueuing the PDC buffer, the format of the PDC buffer is undefined. As the JT1001 controller receives packets, it places the information into the PDC buffer in the following format. HOST software guarantees receive PDC buffers are aligned on a QWORD boundary. The maximum size of a receive PDC buffer is 64 Kbytes. HOST software must guarantee that the maximum size is not exceeded.

Receive PDC buffers begin with either a receive packet header or null header. Following each header, 0 – n bytes of data is written to the PDC buffer. HOST software can determine the type of header by examining the HDTYPE field. If a header has data following it, HOST software can use the HDTYPE or LEN field to determine the offset of the next header in the buffer. Headers within a receive PDC buffer are aligned on a QWORD boundary. The last header in a receive PDC buffer is always the null header. A detailed description of each header type follows.

A receive packet header describes a received packet. For each received packet in a PDC buffer, the JT1001 controller will write a receive packet header followed by the data bytes that constitute the received packet. A single receive PDC buffer may contain multiple receive PDC headers. Figure 5-5 describes the format of the receive packet header and data.

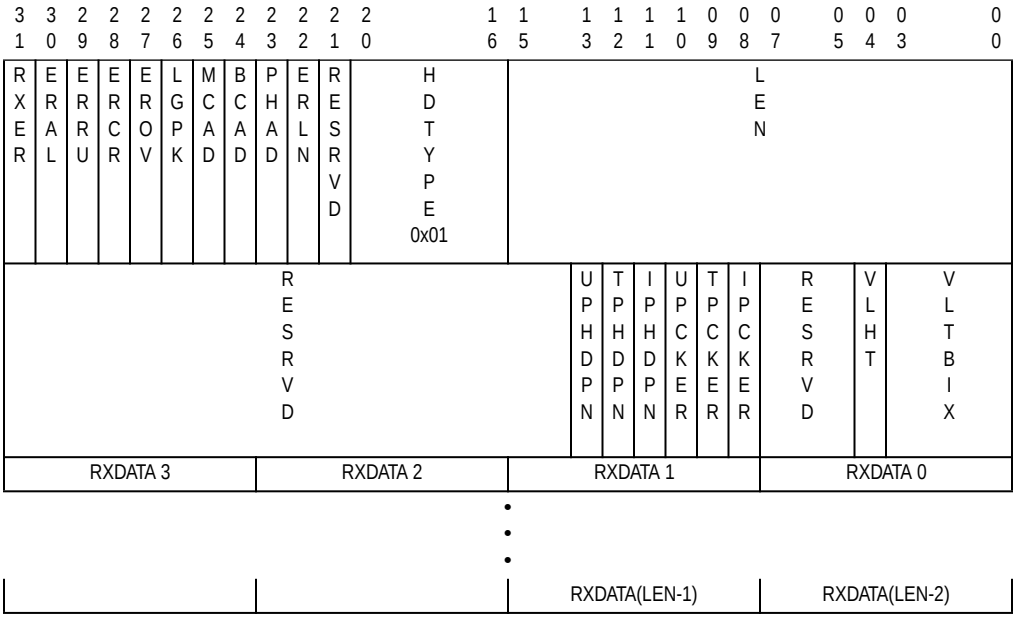


Figure 5-5. PDC Receive Header and Data Format

The definition of the PDC receive header and data fields follow.

LEN — *Length.* Indicates the number of bytes in the received packet. LEN bytes of received data follow the receive packet header.

HDTYPE — *Header Type.* A unique value used to identify the header type. For a receive packet header, the value is 0x1.

PHAD — *Physical Address.* The PHAD bit indicates that the received packet's destination address matches the JT1001 controller's station (MAC) address.

BCAD — *Broadcast Address.* This bit indicates that the received packet's destination address was the broadcast address.

MCAD — *Multicast Address.* When the JT1001 controller multicast address filtering mechanism determines that a packet with a multicast destination address should be passed to the HOST, it sets this bit in the PDL header and transfers the packet to HOST memory.

LGPK — *Large Packet.* By setting this bit, the JT1001 controller indicates that the inbound packet was determined to be larger than the maximum allowable length for an ethernet frame. If the LGPKEN enable bit is clear, the JT1001 controller regards this condition as an error. If the LGPKEN enable bit is set, the JT1001 controller does not regard this condition as an error.

EROV — *Overflow Error.* For PDCs, buffer overflow can only occur when the first packet to be deposited in the PDC buffer is larger than the buffer proper. In this case, the JT1001 controller will deliver as much data as will fit into the available buffer space and set the EROV bit. Any data that does not fit into the buffer will be discarded. When processing a PDC that already contains one or more packets, a packet that does not fit into the remaining buffer space does not cause an overflow error. Instead, the PDC completes (i.e., it is given to the HOST), and the packet in question is delivered into the next available PDC buffer.

ERCR — *CRC Error.* When the JT1001 controller detects that an inbound packet's CRC does not match the computed value, it sets this bit to signal the condition.

ERRU — *Runt Error.* If the JT1001 controller determines that an inbound packet is shorter than the minimum ethernet packet length, it sets the ERRU bit.

ERAL — *Alignment Error.* This bit is set when the JT1001 controller receives a packet that is not an integral number of octets in length.

ERLN — *Length Error.* This bit is set when the JT1001 controller detects that an inbound packet's LLC data is shorter than the length specified in the length/type field of the packet's MAC header.

RXER — *Receive Error.* Whenever an error condition is detected for a receive packet, the error bit in that receive packet header is set to 1. This error bit is the "OR" of the ERLN, EROV, ERCR, ERRU, and ERAL bits. This bit is also set if the LGPKEN bit in *Mode Register – 1* is clear, the PAERPKEN bit in *Mode Register – 1* is set, and the LGPK bit is set.

VLTBIX — *VLAN Table Index.* This field indicates the index of the VLAN TCI Table entry that matched the TCI in the received packet's VLAN tag header. This field only has meaning if the VLHT bit is set.

VLHT — *VLAN Hit.* When set, this bit indicates the received packet contained a VLAN tag header whose TCI matched an entry in the VLAN TCI Table. This bit is set by the JT1001 controller if the VLEN and VLTBEN bits in *Mode Register – 1* are set and the VLAN tag information in the packet matches an entry in the VLAN TCI Table; otherwise this bit will not be set.

IPCKER — *IP Header Checksum Error.* When set, this bit indicates the packet failed the IP header checksum test. The JT1001 controller tests the IP header checksum in packets when the RXIPCKEN bit is set in *Mode Register – 2* and the packet contains an IP header. When clear, the packet either passed the IP header checksum test, did not contain an IP header, or the JT1001 controller's checksum support is disabled.

TPCKER — *TCP Checksum Error.* When set, this bit indicates the packet failed the TCP checksum test. The JT1001 controller tests the TCP checksum in packets when the RXTPCKEN bit is set in *Mode Register – 2* and the packet contains a TCP header and data. When clear, the packet either passed the TCP checksum test, did not contain a TCP header, or the JT1001 controller's checksum support is disabled.

UPCKER — *UDP Checksum Error.* When set, this bit indicates the packet failed the UDP checksum test. The JT1001 controller tests the UDP checksum in packets when the RXUPCKEN bit is set in *Mode Register – 2* and the packet contains a UDP header and data. When clear, the packet either passed the UDP checksum test, did not contain an UDP header, or the JT1001 controller's checksum support is disabled.

IPHDPN — *IP Header Present.* When set, this bit indicates the JT1001 controller found an IP header in the packet. When clear, the packet did not contain an IP header. The value of this bit is valid irrespective of the setting of the RXIPCKEN bit in *Mode Register – 2*.

TPHDPN — *TCP Header Present.* When set, this bit indicates the JT1001 controller found a TCP header in the packet. When clear, the packet did not contain a TCP header. The value of this bit is valid irrespective of the setting of the RXTPCKEN bit in *Mode Register – 2*.

UPHDPN — *UDP Header Present.* When set, this bit indicates the JT1001 controller found a UDP header in the packet. When clear, the packet did not contain a UDP header. The value of this bit is valid irrespective of the setting of the RXUPCKEN bit in *Mode Register – 2*.

RXDATA — *Receive Data.* Data bytes that constitute the received packet.

A null header is used to indicate that no more headers exist in a PDC. Figure 5-6 describes the format of the null header.

31	21	20	16	15	0
RESRVD		HDTYPE 0x00		RESRVD	
See Figure 5-4. PDC Null Header Format. See Figure 5-5. PDC Null Header Format.					

Figure 5-6. PDC Null Header Format

The definition of the PDC null header field follows.

HDTYPE — *Header Type.* A unique value used to identify the header type. For a null header, the value is 0x0.

When an overflow occurs in a PDC buffer, a null header is not deposited into the PDC buffer, since the overflow condition implies that the first packet to be delivered was too big to fit. In particular, this is true even if the null header itself is the source of the overflow; e.g., as may occur when a PDC buffer is just large enough to accommodate an inbound packet but not the null header appended by the JT1001 controller.

CSR 18 INTERRUPT PERIOD REGISTER RESERVED

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TMPE																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	RW	x	TMPE	FFFFFFFFh	**NOTE: The definition of this register is temporary and will be changed in a future revision of the JT1001 controller.** *Timer Period.* The value in this register specifies the number of clock ticks that elapse before the JT1001 controller generates an interrupt if either the DLINRQ or PEINRQ bits are set in the *Command Register* and the Timer Expired Interrupt Mask bit is set in the *Interrupt Mask Register.* The clock operates at the speed of the PCI bus (i.e., 33 MHz or 66 MHz). When the DLINRQ bit is set in the *Command Register,* a single interrupt is generated after the time specified here elapses. If the PEINRQ bit is set in the *Command Register,* an interrupt is generated each time the count elapses. The PEINRQ bit takes precedence over the DLINRQ bit.

CSR 19 TX FIFO PACKET COUNT REGISTER

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16	15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
RESRVD	TXFIPKCN

Bit Field	Type	E^2	Mnemonic	Default Value	Description
15:0	R	x	TXFIPKCN	0	*TX FIFO Packet Count.* The number of packets in the TX FIFO.
31:16	x	x	RESRVD	0	*Reserved.*

CSR 20 TX FIFO LOW WATERMARK REGISTER

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16	15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
RESRVD	TXFILOWM

Bit Field	Type	E^2	Mnemonic	Default Value	Description
15:0	RW	x	TXFILOWM	1000h	*TX FIFO Low Watermark.* A write to this register sets the low water mark. When the number of DWORDs in the TX FIFO becomes equal to the value written into this register, the JT1001 controller will signal a TXWMIN. The actual generation of an interrupt request is governed by the TXWMINMS.
31:16	x	x	RESRVD	0	*Reserved.*

CSR 21 TX FIFO DWORDS FREE REGISTER

31–16	15–0
RESRVD	TXFIDWCN

Bit Field	Type	E²	Mnemonic	Default Value	Description
15:0	R	x	TXFIDWCN	2000h	*Transmit DWORDs Free.* The number of DWORDS free in the TX FIFO.
31:16	x	x	RESRVD	0	*Reserved.*

CSR 22 TX FIFO WRITE REGISTER

31–0
TXFIWT

Bit Field	Type	E²	Mnemonic	Default Value	Description
31:0	W	x	TXFIWT	N/A	*TX FIFO Write.* A write to this register will store the value in the TX FIFO. The write can be performed as a BYTE, WORD, or DWORD operation.

When operating in PIO mode, HOST software accesses the TX FIFO directly via this CSR. Data can be written to the CSR using BYTE, WORD, and DWORD accesses.

The first DWORD written to the FIFO contains the number of bytes in the packet. Subsequent writes to the FIFO contain the packet data itself. Once all packet data has been written to the FIFO, HOST software initiates the transmission by setting the SLMDTXCM bit in the *Command Register*. It is the HOST software's responsibility to enforce minimum and maximum packet sizes. The maximum size packet that can be written into the TX FIFO is 32 Kbytes (including the TX FIFO packet header). HOST software must not write to the TX FIFO when it is full.

Figure 5-7 describes the format and sequence of the writes to the TX FIFO.

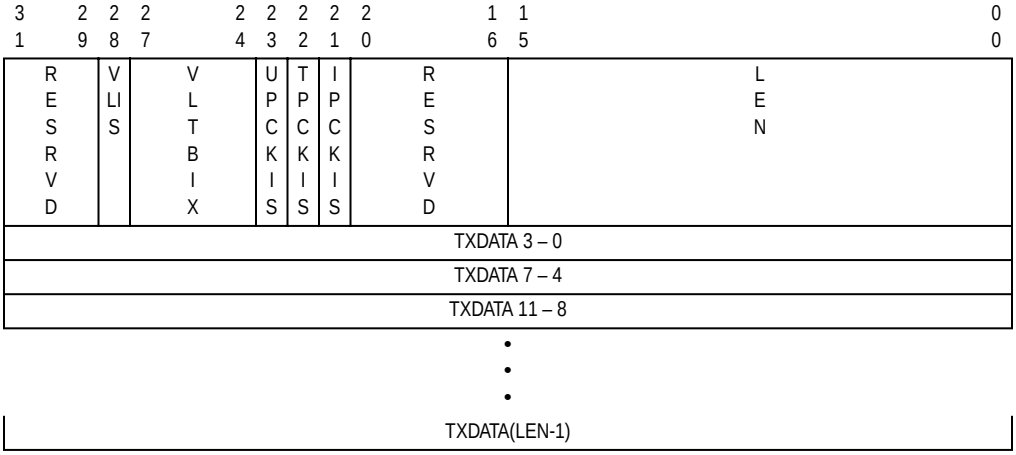


Figure 5-7. PIO Transmit Header and Data Format

The definitions for the PIO transmit header and data fields follow.

LEN — *Length.* Indicates the number of bytes to be transmitted. LEN bytes of transmit data follow the transmit packet header.

IPCKIS — *IP Header Checksum Insert.* This bit allows HOST software to request the IP header checksum be inserted into the packet. Setting this bit causes the JT1001 controller to calculate and insert the IP header checksum into the packet that contains an IP header. If the packet does not contain an IP header, the JT1001 controller does not calculate or insert the checksum.

TPCKIS — *TCP Checksum Insert*. This bit allows HOST software to request the TCP checksum be inserted into the packet. Setting this bit causes the JT1001 controller to calculate and insert the TCP checksum into a packet that contains a TCP header. If the packet does not contain a TCP header, the JT1001 controller does not calculate or insert the checksum.

UPCKIS — *UDP Checksum Insert.* This bit allows HOST software to request the UDP checksum be inserted into the packet. Setting this bit causes the JT1001 controller to calculate and insert the UDP checksum into the packet that contains a UDP header. If the packet does not contain a UDP header, the JT1001 controller does not calculate or insert the checksum.

VLTBIX — *VLAN TCI Table Index*. This field is an index into the VLAN TCI Table. The JT1001 controller uses the TCI information at this index to construct a VLAN tag header.

VLIS — *VLAN Insert Tag*. Setting this bit causes the JT1001 controller to construct and insert a VLAN tag header into the packet prior to its transmission. The JT1001 controller constructs the VLAN tag header using the TCI at index VLTBIX in the VLAN TCI Table.

TXDATA — *Transmit Data.* Data bytes that constitute the packet to be transmitted.

CSR 23 RESERVED

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RESRVD																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	x	x	RESRVD	0	*Reserved.*

CSR 24 RX FIFO READ REGISTER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RXFIRD																RXFIRD								RXFIRD							

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	R	x	RXFIRD	N/A	*RX FIFO Read.* A read from this register will extract the next available value in the RX FIFO. A read when the RX FIFO is empty is undefined and yields invalid data. The read can be performed as a BYTE, WORD, or DWORD operation.

When operating in PIO mode, HOST software accesses the RX FIFO directly via the *RX FIFO Read Register.* Each read of this CSR retrieves one DWORD of data from the RX FIFO.

The first DWORD read from the FIFO returns the reception status and packet length. The second DWORD read from the FIFO returns VLAN tag information. Subsequent reads to the FIFO return the packet data itself. The maximum size packet that can be read from the RX FIFO is 64 Kbytes (including the RX FIFO packet header). NOTE: Dropped packets will not appear in the FIFO as an errored packet. If a packet is dropped, the JT1001 controller will simply increment the *Dropped Packet Count Register*.

Figure 5-8 describes the format of a packet in the RX FIFO.

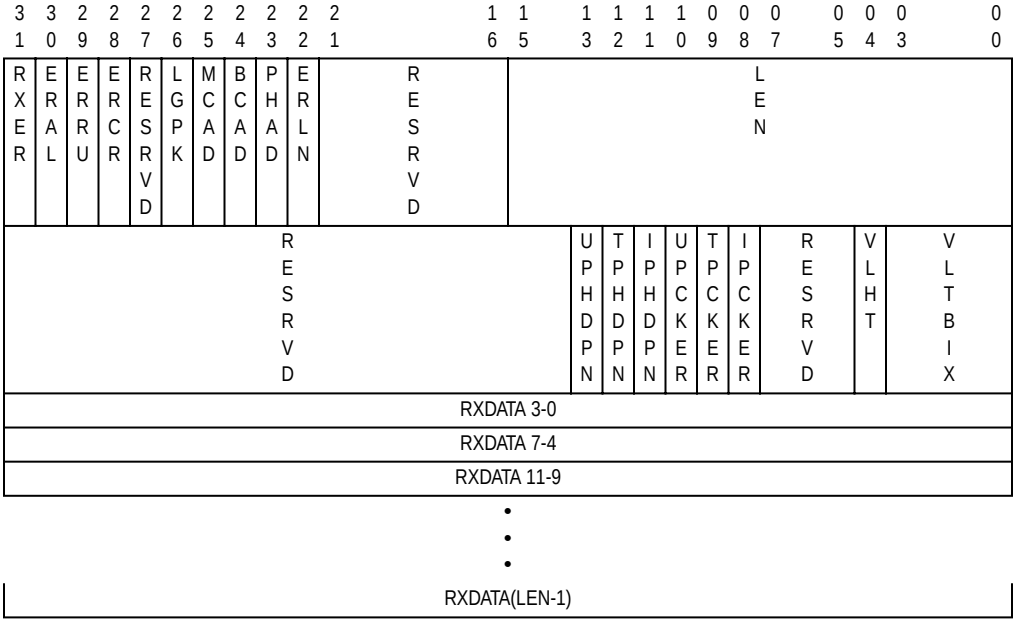


Figure 5-8. PIO Receive Header and Data Format

The definitions for the PIO receive header and data fields follow.

LEN — *Length.* Indicates the number of bytes actually deposited into FIFO for the received packet.

PHAD — *Physical Address.* The PHAD bit indicates that the received packet's destination address matches the JT1001 controller's station (MAC) address.

BCAD — *Broadcast Address.* This bit indicates that the received packet's destination address was the broadcast address.

MCAD — *Multicast Address.* The JT1001 controller sets this bit to indicate the received packet met the following conditions: (1) the destination address is a multicast address, and (2) the multicast address hashing algorithm generated a bit that matches a bit set in the *Multicast Hash Table Register*.

LGPK — *Large Packet.* By setting this bit, the JT1001 controller indicates that the inbound packet was determined to be larger than the maximum allowable length for an ethernet frame. If the LGPKEN enable bit is clear, the JT1001 controller regards this condition as an error. If the LGPKEN enable bit is set, the JT1001 controller does not regard this condition as an error.

ERCR — *CRC Error.* When the JT1001 controller detects that an inbound packet's CRC does not match the computed value, it sets this bit to signal the condition.

ERRU — *Runt Error.* If the JT1001 controller determines that an inbound packet is shorter than the minimum ethernet packet length, it sets the ERRU bit.

ERAL — *Alignment Error.* This bit is set when the JT1001 controller receives a packet that is not an integral number of octets in length.

ERLN — *Length Error.* This bit is set when the JT1001 controller detects that an inbound packet's LLC data is shorter than the length specified in the length/type field of the packet's MAC header.

RXER — *Receive Error.* The JT1001 controller sets this bit when an error condition is detected for a received packet. This error bit is the "OR" of the ERLN, ERCR, ERRU, and ERAL bits. This bit is also set if the LGPKEN bit in *Mode Register – 1* is clear, the PAERPKEN bit in *Mode Register – 1* is set, and the LGPK bit is set.

VLTBIX — *VLAN Table Index*. This field indicates the index of the VLAN TCI Table entry that matched the TCI in the received packet's VLAN tag header. This field has meaning only if the VLHT bit is set.

VLHT — *VLAN Hit*. When set, this bit indicates the received packet contained a VLAN tag header whose TCI matched an entry in the VLAN TCI Table. This bit is set by the JT1001 controller if the VLEN and VLTBEN bits in *Mode Register – 1* are set and the VLAN tag information in the packet matches an entry in the VLAN TCI Table; otherwise this bit will not be set.

IPCKER — *IP Header Checksum Error*. When set, this bit indicates that the packet failed the IP header checksum test. The JT1001 controller tests the IP header checksum in packets when the RXIPCKEN bit is set in *Mode Register – 2* and the packet contains an IP header. When clear, the packet either passed the IP header checksum test, did not contain an IP header, or the JT1001 controller's checksum support is disabled.

TPCKER — *TCP Checksum Error*. When set, this bit indicates that the packet failed the TCP checksum test. The JT1001 controller tests the TCP checksum in packets when the RXTPCKEN bit is set in *Mode Register – 2* and the packet contains a TCP header and data. When clear, the packet either passed the TCP checksum test, did not contain a TCP header, or the JT1001 controller's checksum support is disabled.

UPCKER — *UDP Checksum Error*. When set, this bit indicates that the packet failed the UDP checksum test. The JT1001 controller tests the UDP checksum in packets when the RXUPCKEN bit is set in *Mode Register – 2* and the packet contains a UDP header and data. When clear, the packet either passed the UDP checksum test, did not contain an UDP header, or the JT1001 controller's checksum support is disabled.

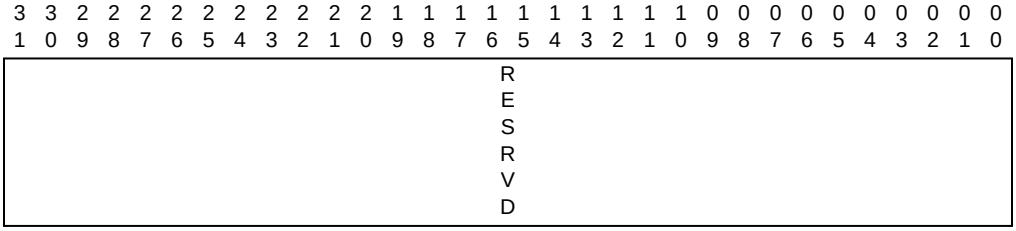
IPHDPN — *IP Header Present.* When set, this bit indicates that the JT1001 controller found an IP header in the packet. When clear, the packet did not contain an IP header. The value of this bit is valid irrespective of the setting of the RXIPCKEN bit in *Mode Register – 2*.

TPHDPN — *TCP Header Present.* When set, this bit indicates that the JT1001 controller found a TCP header in the packet. When clear, the packet did not contain a TCP header. The value of this bit is valid irrespective of the setting of the RXTPCKEN bit in *Mode Register – 2*.

UPHDPN — *UDP Header Present.* When set, this bit indicates that the JT1001 controller found an UDP header in the packet. When clear, the packet did not contain an UDP header. The value of this bit is valid irrespective of the setting of the RXUPCKEN bit in *Mode Register – 2*.

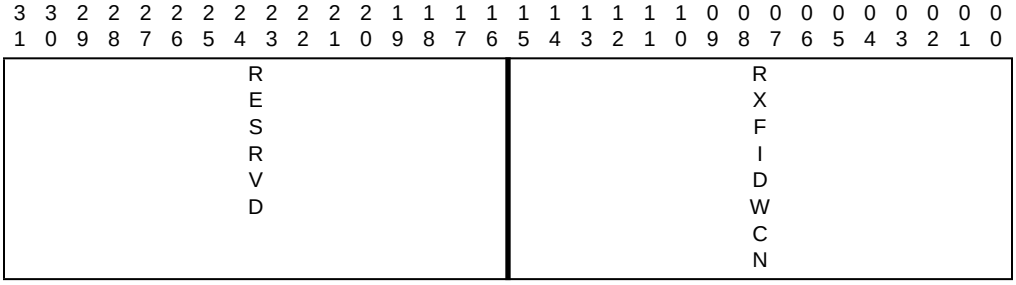
RXDATA — *Received Data.* Data bytes that constitute the received packet. If the RXDATA does not end on a QWORD boundary, HOST software will issue a RX FIFO skip packet command to the JT1001 controller. Alternatively, HOST software can also perform one more read of the RX FIFO and discard the data.

CSR 25 RESERVED



Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	x	x	RESRVD	0	*Reserved.*

CSR 24 RX FIFO DWORD COUNT REGISTER



Bit Field	Type	E^2	Mnemonic	Default Value	Description
15:0	R	x	RXFIDWCN	0	*RX FIFO DWORD Count.* The number of DWORDS currently consumed by received packets in the RX FIFO.
31:16	x	x	RESRVD	x	*Reserved.*

CSR 27 RX FIFO HIGH WATERMARK REGISTER

31:16	15:0
RESRVD	RXFIHIWM

Bit Field	Type	E²	Mnemonic	Default Value	Description
15:0	RW	x	RXFIHIWM	2000h	*RX FIFO High Watermark.* A write to this register sets the high water mark for the RX FIFO. When the number of DWORDs in the RX FIFO becomes equal to the value written into this register, the JT1001 controller will signal a RXWMIN. The actual generation of an interrupt request is governed by the RXWMINMS.
31:16	x	x	RESRVD	x	*Reserved.*

CSR 28 RX FIFO PACKET COUNT REGISTER

31:16	15:0
RESRVD	RXFIPKCN

Bit Field	Type	E²	Mnemonic	Default Value	Description
15:0	R	x	RXFIPKCN	0	*RX FIFO Packet Count.* The number of packets in the RX FIFO.
31:16	x	x	RESRVD	x	*Reserved.*

CSR 29 COMMAND REGISTER

31:16	15:8	7	6	5	4	3	2	1	0
RESRVD	RESRVD	RESRVD	TMEN1	TMEN0	PEINRQ	DLINRQ	RXFISKPK	SLMDTXCM	SERECL

Bit Field	Type	E^2	Mnemonic	Default Value	Description
0	W	x	SERECL	N/A	*Set/Reset Control.* Set/reset control bit for bits[7:1].
1	WA	x	SLMDTXCM	0	*Slave Mode Transmit Command.* HOST software sets this bit to initiate the transmission of a packet it has placed in the TX FIFO. Prior to setting this bit, HOST software must write all the packet's PIO transmit header and packet data to the FIFO using the *TX FIFO Write Register.*
2	WA	x	RXFISKPK	0	*RX FIFO Skip Packet.* Setting this bit causes the current packet in the RX FIFO to be discarded. The JT1001 controller advances the RX FIFO's current packet pointer to the next available packet and decrements the *RXFIPKCN Register.*
3	WA	x	DLINRQ	0	*Delayed Interrupt Request.* Setting this bit causes the JT1001 controller to start a countdown timer. Upon expiration of the timer, the JT1001 controller clears the DLINRQ bit and generates an interrupt if the TMEXMS bit in the *Interrupt Mask Register* is set. The initial value of the countdown timer is determined by the value in the *Interrupt Period Register.* If HOST software sets the DLINRQ bit again before the countdown timer expires, the JT1001 controller will reload the counter with the current value of the *Interrupt Period Register.* If HOST software resets the DLINRQ bit before the countdown timer expires, the JT1001 controller will cancel the timer and no interrupt occurs.
4	RW	x	PEINRQ	0	*Periodic Interrupt Request.* Setting this bit causes the JT1001 controller to start a countdown timer. The duration of the timer is the value specified in the *Interrupt Period Register.* Upon expiration of the timer, the JT1001 controller performs the following actions: • Checks the state of the TMEXMS bits in the *Interrupt Mask Register*. If the TMEXMS bit is set, the JT1001 controller generates an interrupt. • Checks the state of the PEINRQ bit. If it is still set, the JT1001 controller reloads the timer with the value in the *Interrupt Mask Register* and restarts the timer. The JT1001 controller continues in this cycle until the HOST software resets the PEINRQ bit. If the HOST software resets the PEINRQ bit before the countdown timer expires, the JT1001 controller cancels the timer and no more periodic timer interrupts occur.

Bit Field	Type	E^2	Mnemonic	Default Value	Description
5	RW	x	TMEN0	0	*Timer 0 Enable.* Setting this bit starts Timer 0 ticking. The timer ticks at a rate of 25 MHz. Each tick causes the value in the *Timer 0 Count Register* to be incremented. Clearing this bit causes the timer to stop ticking and, therefore, the value in the *Timer 0 Count Register* to stop incrementing.
6	RW	x	TMEN1	0	*Timer 1 Enable.* Setting this bit starts Timer 1 ticking. The timer ticks at a rate of 25 MHz. Each tick causes the value in the *Timer 1 Count Register* to be incremented. Clearing this bit causes the timer to stop ticking and, therefore, the value in the *Timer 1 Count Register* to stop incrementing.
7	x	x	RESRVD	0	*Reserved.*
15:8	x	x	RESRVD	0	*Reserved.*
31:16	x	x	RESRVD	0	*Reserved.*

CSR 30 INTERRUPT MASK REGISTER

This register governs the JT1001 controller's ability to generate interrupts.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RESRVD								RESRVD			TMMS1	TMMS0	USPIMS1	USPIMS0	SERECL	RESRVD	RXMGPKMS	PHLASTMS	RXPDMS	RXMS	RXFIWMMS	RXCMEMMS	SERECL	INENMS	TMEXMS	RESRVD		TXDMDNMS	TXFIWMMS	TXCMEMMS	SERECL

Bit Field	Type	E^2	Mnemonic	Default Value	Description
0	W	x	SERECL	N/A	*Set/Reset Control.* Set/reset control bit for bits[7:1].
1	RW	x	TXCMEMMS	0	*TX Command FIFO Empty Interrupt Mask.*
2	RW	x	TXFIWMMS	0	*TX FIFO Watermark Interrupt Mask.*
3	RW	x	TXDMDNMS	0	*Transmit DMA Done Interrupt Mask.*
5:4	x	x	RESRVD	0	*Reserved.*
6	RW	x	TMEXMS	0	*Timer Expired Interrupt Mask.* When set, an interrupt will occur when the single shot or periodic timer has expired. This timer is started when the setting of either the DLINRQ or PEINRQ bits in the *Command Register* has expired. The duration of the timer is the determined by the value specified in the *Interrupt Period Register.*

Bit Field	Type	E^2	Mnemonic	Default Value	Description
7	RW	x	INENMS	0	*Interrupt Enable Mask.* This bit is the master interrupt enable bit that enables/disables the JT1001 controller's ability to generate an interrupt. When set, the JT1001 controller will generate an interrupt whenever an event bit (bits 23:0) in the *Event Status Register* is set and the event bit's corresponding mask bit is set in this register. For example, if the INENMS bit is set *(*bit 6 is set in the *Event Status Register* and in the *Interrupt Mask Register)*, the JT1001 controller generates an interrupt. When the INENMS bit is clear, the JT1001 controller's ability to generate an interrupt is disabled. NOTE: Clearing the INENMS bit does NOT prevent the JT1001 controller from setting event bits [23:0] in the *Event Status Register.* Clearing the INENMS bit merely prevents the JT1001 controller from interrupting the HOST. If any of the *Event Status Register* bits [23:0] are set and the event bit's corresponding mask bit is set when HOST software sets INENMS bit, the JT1001 controller will interrupt the HOST immediately.
8	W	x	SERECL	0	*Set/Reset Control.* Set/reset control bit for bits [15:9].
9	RW	x	RXCMEMMS	0	*RX Command FIFO Empty Interrupt Mask.*
10	RW	x	RXFIWMMS	0	*RX FIFO Watermark Interrupt Mask.*
11	RW	x	RXMS	0	*Receive Interrupt Mask.* This bit enables the JT1001 controller to generate interrupts whenever the JT1001 controller has placed a complete packet into the RX FIFO.
12	RW	x	RXPDMS	0	*Receive PDL/PDC Interrupt Mask.* When set, this bit enables per PDC/PDL interrupts as requested in the flags field of the PDC/PDL descriptors or commands. The interrupts are generated when the JT1001 controller has completely transferred the packet data to HOST buffers.
13	RW	x	PHLASTMS	0	*Physical Layer Status Interrupt Mask.* When set, the JT1001 controller generates an interrupt when a PHY status change occurs.
14	RW	x	RXMGPKMS	0	*Receive Magic Packet Mask.* When set, reception of a Magic Packet data sequence will generate an interrupt.
15	x	x	RESRVD	0	*Reserved.*
16	W	x	SERECL	0	*Set/Reset Control.* Set/reset control bit for bits [23:17].
17	RW	x	USPIMS0	0	*User Pin0 Interrupt Mask.* This bit enables/ disables the generation of an interrupt based upon the state of User Pin0. If this bit is set, and the User Pin0 transitions from a low to high state, the JT1001 controller generates an interrupt.
18	RW	x	USPIMS1	0	*User Pin1 Interrupt Mask.* This bit enables/ disables the generation of an interrupt based upon the state of User Pin1. If this bit is set, and the User Pin1 transitions from a low to high state, the JT1001 controller generates an interrupt.

Bit Field	Type	E^2	Mnemonic	Default Value	Description
19	RW	x	TMMS0	0	*Timer 0 Interrupt Mask.* This bit enables/disables the generation of an interrupt when the value of the *Timer 0 Counter Register* equals the value of the *Timer 0 Interrupt Trigger Register.*
20	RW	x	TMMS1	0	*Timer 1 Interrupt Mask.* This bit enables/disables the generation of an interrupt when the value of the *Timer 1 Counter Register* equals the value of the *Timer 1 Interrupt Trigger Register.*
31:21	x	x	RESRVD	0	*Reserved.*

CSR 31 RESERVED

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RESRVD																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	x	x	RESRVD	0	*Reserved.*

CSR 32 EVENT STATUS REGISTER

The *Event Status Register* indicates the events that have been detected by the device. When an event is detected, the corresponding bit in this register is set. If both the INENMS bit and the event's corresponding mask bit are set in the *Interrupt Mask Register,* then the occurrence of the event causes the device to signal an interrupt. Following a read of *any* of the event status bits in this register, the JT1001 controller automatically resets *all* the event status bits. Moreover, the JT1001 controller automatically resets the INENMS bit in the *Interrupt Mask Register* if an interrupt is pending (i.e., the JT1001 controller's interrupt line is active). This automatic clearing of the INENMS bit disables the JT1001 controller's ability to generate further interrupts. To re-enable JT1001 controller interrupts, the HOST software must set the INENMS bit in the *Interrupt Mask Register.* When the HOST software sets the INENMS bit, the JT1001 controller will immediately signal an interrupt for any pending events; i.e., events having occurred after the last read of the *Event Status Register*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RXDMDNCN								RESRVD			TMIN1	TMIN0	USPIIN1	USPIIN0	RESRVD	RESRVD	RXMGPKIN	PHLASTIN	RXPDIN	RXIN	RXFIWMIN	RXCMEMIN	RESRVD	RESRVD	TMEXIN	RESRVD		TXDMDNIN	TXFIWMIN	TXCMEMIN	RESRVD

Bit Field	Type	E[2]	Mnemonic	Default Value	Description
0	x	x	RESRVD	N/A	*Reserved.*
1	RC	x	TXCMEMIN	0	*Transmit Command FIFO Empty Interrupt.* This interrupt signals that the transmit command FIFO is empty. This bit is set when the transmit command count rises above 0 and then returns to 0.
2	RC	x	TXFIWMIN	0	*TX FIFO Watermark Interrupt.* This interrupt is asserted when the number of DWORDS in the TX FIFO becomes equal to the number of DWORDS specified in the *TXFIWM Register* due to a read by the MAC.
3	RC	x	TXDMDNIN	0	*Transmit DMA Done Interrupt.* When set, this bit indicates that the JT1001 controller has transferred a packet with the DMDINRQ flag set in the PDL or PDC command field. If multiple packets have been transferred, as can happen with a PDC, the interrupt is signaled after the final packet has been copied to the JT1001 controller.
5:4	RC	x	RESRVD	0	*Reserved.*
6	RC	x	TMEXIN	0	*Timer Expired Interrupt.*
7	RC	x	RESRVD	0	*Reserved.*
8	RC	x	RESRVD	0	*Reserved.*
9	RC	x	RXCMEMIN	0	*Receive Command FIFO Empty Interrupt.* This interrupt signals that the receive command FIFO is empty. The JT1001 controller asserts this signal immediately after the last receive PDC/PDL is extracted from the receive command FIFO. This bit is set when the receive command count rises above 0 and then returns to 0.
10	RC	x	RXFIWMIN	0	*RX FIFO Watermark Interrupt.* This interrupt is asserted when the number of DWORDS in the RX FIFO becomes equal to the number of DWORDS specified in the *RXFIWM Register* due to a write by the MAC.
11	RC	x	RXIN	0	*Receive Interrupt.* The JT1001 controller generates receive interrupts whenever a complete packet is available in the RX FIFO. Note that RXIN interrupts override RXPDIN interrupts. In other words, when the RXMS bit is set, an RXIN interrupt is generated for each packet that is received by the JT1001 controller irrespective of how the RXINRQ bit is set in the receive PDCs and PDLs.

Bit Field	Type	E^2	Mnemonic	Default Value	Description
12	RC	x	RXPDIN	0	*Receive PDC/PDL Interrupt.* When the *Interrupt Mask Register* RXPDMS bit is set to 1, the JT1001 controller generates receive interrupts for each packet received into a PDC or PDL style buffer that has the RXINRQ bit set. The RXPDIN interrupt is asserted only after the JT1001 controller has transferred the complete packet (or group of packets in PDC mode) to HOST memory. Note that if RXIN interrupts are also enabled (InterruptMaskRegister.RXMS=1), then each received packet generates an interrupt (InterruptStatusRegister.RXIN=1) irrespective of the state of the RXINRQ bit in receive PDCs or PDLs.
13	x	x	PHLASTIN	0	*Physical Layer Status Interrupt.* When set, this bit indicates the JT1001 controller has detected a PHY status change. HOST software can obtain the current PHY status via the *G/MII PHY Access Register.*
14	RC	x	RXMGPKIN	0	*Receive Magic Packet Interrupt.* When set, this bit indicates that a Magic Packet data sequence was received.
15	RC	x	RESRVD	0	*Reserved*.
16	RC	x	RESRVD	0	*Reserved*.
17	RC	x	USPIIN0	0	*User Pin0 Interrupt.* An interrupt occurs when the User Pin0 transitions from a low to high state.
18	RC	x	USPIIN1	0	*User Pin1 Interrupt.* An interrupt occurs when the User Pin1 transitions from a low to high state.
19	RC	x	TMIN0	0	*Timer 0 Interrupt.* An interrupt occurs when the value of the *Timer 0 Count Register* equals the value of the *Timer 0 Interrupt Trigger Register*.
20	RC	x	TMIN1	0	*Timer 1 Interrupt.* An interrupt occurs when the value of the *Timer 1 Count Register* equals the value of the *Timer 1 Interrupt Trigger Register*.
23:21	RC	x	RESRVD	0	*Reserved*.
31:24	RC	x	RXDMDNCN	0	*Receive DMA Done Count.* The count of receive PDL/PDCs that the JT1001 controller has completed processing since the last read of this field. The receive command queue can hold 31 commands. HOST software uses this field to determine how many receive PDL/PDC buffers have been filled with receive data by the JT1001 controller. The JT1001 controller resets the count to zero after each read. NOTE: This field can be accessed as a BYTE, WORD, or DWORD. A BYTE read of this field has no affect on the state of other fields in the CSR. A BYTE read of this field does NOT disable the JT1001 controller's ability to generate interrupts. NOTE: This field is aliased from the *Command Status Register.* Whether the count is read via the *Command Status Register* or the *Event Status Register,* the JT1001 controller will reset the count after the read.

CSR 33 RESERVED

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RESRVD																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	x	x	RESRVD	0	*Reserved.*

CSR 34 MULTICAST HASH TABLE REGISTER LSD

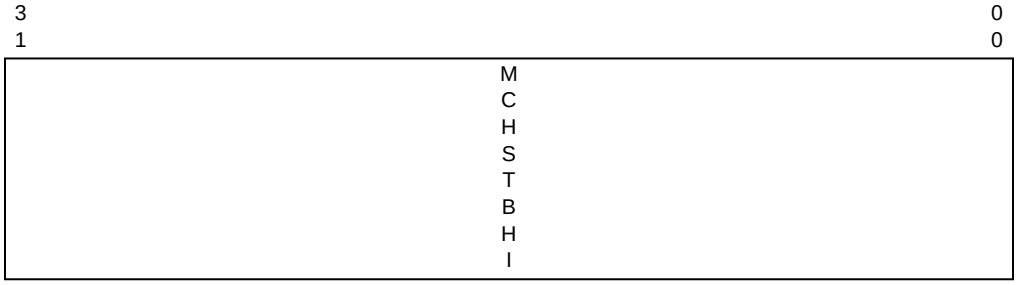
When enabling a multicast address, the driver computes the hash value and makes the JT1001 controller aware of the new address by writing a new hash value into MCHSTBLO or MCHSTBHI.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
MCHSTBLO																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	W	x	MCHSTBLO	0	*Multicast Hash Table Low.* The Multicast Hash Table is set by the driver to indicate to the JT1001 controller which multicast addresses are acceptable to the HOST. The hashing algorithm is an imperfect filter. Consequently, the HOST must ultimately examine inbound packets with multicast destination addresses to determine if they are indeed intended for the recipient HOST. The Multicast Address Table is 64 bits wide. Bits 31 through 0 of the table are maintained in this register.

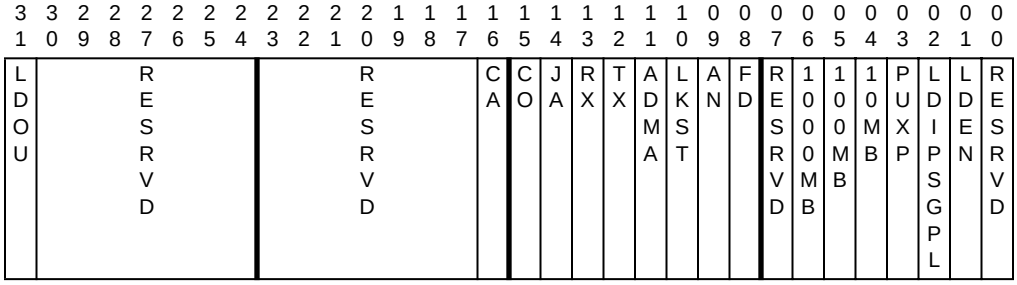
CSR 35 MULTICAST HASH TABLE REGISTER MSD

When enabling a multicast address, the driver computes the hash value and makes the JT1001 controller aware of the new address by writing a new hash value into MCHSTBLO or MCHSTBHI.



Bit Field	Type	E²	Mnemonic	Default Value	Description
31:0	W	x	MCHSTBHI	0	*Multicast Hash Table High.* Bits 63 through 31 of the table are maintained in this register.

CSR 36 LED 0 CONFIGURATION REGISTER



Bit Field	Type	E²	Mnemonic	Default Value	Description
0	x	x	RESRVD	0	*Reserved.*
1	RW	√	LDEN	1	*LED Enable.* Allows the LED to be turned off without upsetting the programming of the *LED Configuration Register.* This bit permits the LED configuration to be taken directly from EEPROM without driver intervention. The driver need only enable the LED if it is disabled.
2	RW	√	LDIPSGPL	1	*LED Input Signal Polarity.* Inverts the input signal to the LED.
3	RW	√	PUXP	1	*Pulse Expander.* Stretches the time that the LED is on (or off) such that it can be easily perceived visually.
4	RW	√	10MB	0	*10 Megabit.* Indicates that the JT1001 controller is configured for 10 Mb operation. When this bit is set and the 10MB bit in the *LED Signal Latch Register* is set, the signal sent to the LED will oscillate at 1 Hz.
5	RW	√	100MB	0	*100 Megabit.* Indicates that the JT1001 controller is configured for 100 Mb operation. When this bit is set and the 100MB bit in the *LED Signal Latch Register* is set, the signal sent to the LED will oscillate at 4 Hz.
6	RW	√	1000MB	0	*1000 Megabit.* Indicates that the JT1001 controller is configured for 1000 Mb operation.
7	x	x	RESRVD	0	*Reserved.*
8	RW	√	FD	0	*Full-Duplex.* Indicates that full-duplex operation is enabled.

Bit Field	Type	E^2	Mnemonic	Default Value	Description
9	RW	√	AN	0	*Auto Negotiating.* Indicates that auto negotiation is in progress.
10	RW	√	LKST	1	*Link State.* Indicates whether the link is functional or nonfunctional.
11	RW	√	ADMA	0	*Address Match.* Indicates that an address match with the JT1001 controller's physical address has been detected.
12	RW	√	TX	0	*Transmit.* Indicates that the JT1001 controller is transmitting a frame.
13	RW	√	RX	0	*Receive.* Indicates that the JT1001 controller is receiving a frame.
14	RW	√	JA	0	*Jabber.* Indicates that the JT1001 controller has detected a jabbering station.
15	RW	√	CO	0	*Collision.* This bit indicates that the JT1001 controller is transmitting and receiving data simultaneously. In half-duplex mode, transmitting and receiving data simultaneously is an error condition known as a collision. In full-duplex mode, transmitting and receiving data simultaneously is not an error condition. This bit should not be set when operating in full-duplex mode.
16	RW	√	CA	0	*Carrier Sense.* Indicates PHY has sensed a CARRIER.
30:17	x	x	RESRVD	0	*Reserved.*
31	R	x	LDOU	x	*LED Out.* The signal sent to the LED is routed to this bit as well.

CSR 37 LED 1 CONFIGURATION REGISTER

See *LED 0 Configuration Register* for a detailed description of the programming.

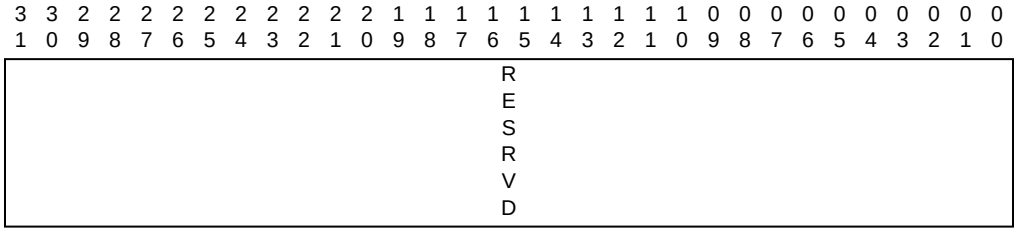
CSR 38 LED 2 CONFIGURATION REGISTER

See *LED 0 Configuration Register* for a detailed description of the programming.

CSR 39 LED 3 CONFIGURATION REGISTER

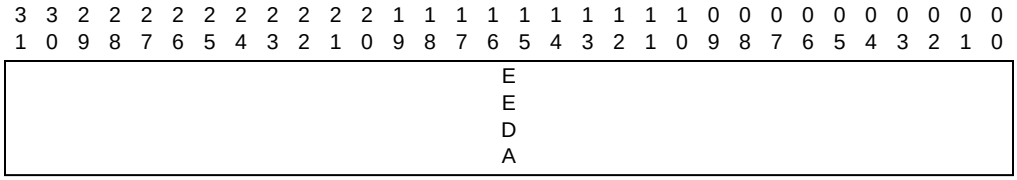
See *LED 0 Configuration Register* for a detailed description of the programming.

CSR 40 RESERVED



Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	R	x	RESRVD	0	*Reserved.*

CSR 41 EEPROM DATA REGISTER



Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	RW	x	EEDA	0	*EEPROM Data Register.* This is the data register used when performing single accesses to EEPROM. When reading from EEPROM, the JT1001 controller places value read from EEPROM in this register. HOST software must not read this register until after the JT1001 controller has cleared the EESI bit. When writing to EEPROM, this register contains the value to be written. HOST software must set the value in this register prior to setting the EESI bit. HOST software must not change this value until after the JT1001 controller has cleared the EESI bit.

CSR 42 LAN PHYSICAL ADDRESS REGISTER LSD

The LAN physical address registers are programmed with the MAC address of the JT1001 controller. The JT1001 controller will accept all frames that have a destination MAC address (OUI) matching the one programmed into this CSR when the unicast enable (UCEN) bit is set in *Mode Register – 1* (CSR 0). When UCEN is reset, the JT1001 controller will not accept *any* unicast frames unless the promiscuous mode enable (POEN) bit is set, in which case *all* frames are accepted.

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 09 08	07 06 05 04 03 02 01 00
PHAD3	PHAD2	PHAD1	PHAD0

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	RW	√	PHAD0-3	0083E000h	*Physical Address 0 – 3.* When programming the JT1001 controller'S MAC address, bytes 0 through 3 of the 6-byte address are written to this register. For purposes of this discussion, byte 0 is the first byte of the OUI as transmitted on the physical medium. The example below further clarifies the point. The JT1001 controller uses the DWORD programmed into this register along with the WORD programmed into CSR 43 to select unicast frames specifically directed at the JT1001 controller.

Jato Technologies' OUI is: 00-E0-83. A sample MAC address based on this OUI is:

00-E0-83-01-02-03

The notation above is frequently referred to as canonical format. In canonical format, the leftmost hexadecimal value (00 in the example) is transmitted first. The hexadecimal value immediately to its right (E0 in the example) is transmitted next, and so on. Each hexadecimal value represents a byte where bit 0 has the value 2^0, bit 1 has the value 2^1, etc., through bit 7. For purposes of this example, the leftmost byte is referred to as byte 0 (PHAD0 in the CSR diagram above) and the rightmost byte is referred to as byte 5.

When bytes 0 through 3 of the physical address are written to this register (CSR 42), byte 0 is written to bits 0 through 7, byte 1 is written to bits 8 through 15, etc. The result is as follows:

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 09 08	07 06 05 04 03 02 01 00
01	83	E0	00

The remaining 2 bytes of the MAC address are deposited into the *LAN Physical Address Register* MSW (CSR 43) as follows:

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 09 08	07 06 05 04 03 02 01 00
Reserved	Reserved	03	02

CSR 43 LAN PHYSICAL ADDRESS REGISTER MSW

31–16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESRVD	PHAD5	PHAD4														

Bit Field	Type	E^2	Mnemonic	Default Value	Description
15:0	RW	√	PHAD 4 – 5	0100h	*Physical Address 4 – 5.* The JT1001 controller's OUI. The most significant WORD of the OUI is programmed in this register.
31:16	RW	x	RESRVD	x	*Reserved.*

CSR 44 G/MII PHY ACCESS REGISTER

31–16	15	14–13	12–8	7	6–5	4–0
GMDA	GMST	RESRVD	GMPHAD	GMCM	RESRVD	GMRRIX

Bit Field	Type	E^2	Mnemonic	Default Value	Description
4:0	RW	x	GMRRIX	0	*G/MII Register Index.* Index of the PHY register to be targeted by the I/O operation.
6:5	x	x	RESRVD	0	*Reserved.*
7	RW	x	GMCM	0	*G/MII Command.* If GMCM = 0, a read of the PHY register specified by GMRRIX is performed. The result is stored in the GMDA. If GMCM = 1, a write of the value in the GMDA is written to the PHY register specified by GMRRIX.
12:8	RW	x	GMPHAD	0	*G/MII Physical Address.* Physical address of the PHY device to which I/O is to be performed.
14:13	x	x	RESRVD	0	*Reserved.*
15	RA	x	GMST	0	*G/MII Command Status.* HOST software polls this bit to determine when a command to the PHY has completed. The JT1001 controller sets this bit when a write to the *G/MII PHY Access Register* occurs. The JT1001 controller will clear this bit when it has completed the I/O operation with the PHY.
31:16	RW	x	GMDA	0	*G/MII Data.* If GMCM = 0, GMDA contains the value read from the PHY. The value will be valid after the GMST bit indicates the read operation has completed. If GMCM = 1, GMDA contains the value to be written to the PHY. GMDA must not be altered by HOST software until the GMST bit indicates the write operation has completed.

HOST software uses the *G/MII PHY Access Register* to access the PHY's status and control registers. Prior to forcing the PHY to renegotiate with its link partner, it is the responsibility of HOST software to quiesce packet transmission and reception by clearing the *Transmit Enable* and *Receive Enable* bits in *Mode Register – 1.*

CSR 45 G/MII MODE REGISTER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RPEN	GMRPP	RESRVD																				GMPCEN	GMIFPR	RESRVD					GMFD	GMWRSP	

Bit Field	Type	E[2]	Mnemonic	Default Value	Description
1:0	RW	x	GMWRSP	10	*G/MII Wire Speed.* HOST software writes these bits to select between the following wire speeds: **GMWRSP[1] / GMWRSP[0] / Line Rate** 1 / 1 / Reserved 1 / 0 / 1000 Mbps 0 / 1 / 100 Mbps 0 / 0 / 10 Mbps HOST software sets this field to match the wire speed at which the PHY is currently operating. HOST software determines the current PHY setting using the *G/MII PHY Access Register.*
2	RW	x	GMFD	1	*G/MII Full-Duplex Mode.* HOST software writes this bit to select between full-duplex mode and half-duplex mode. If GMFD = 1, then full-duplex mode is selected. If GMFD = 0, then half-duplex mode is selected. HOST software sets this field to match the wire speed at which the PHY is currently operating. HOST software determines the current PHY setting using the *G/MII PHY Access Register.*
7:3	RW	x	RESRVD	0	*Reserved.*
8	RW	x	GMIFPR	x	*G/MII Interface Protocol.* This bit selects the interface protocol to use (TBI or G/MII). If GMIFPR is set to logic 1, then the TBI interface protocol is used to connect to a SERDES PHY device. If GMIFPR is cleared to logic 0, then the G/MII interface protocol is used to connect to a G/MII PHY device. Upon reset, this bit is set to the value on the PCS_EN pin.
9	R	x	GMPCEN	x	*G/MII PCS Enhance.* This bit indicates whether the external PHY device needs to be enhanced by the internal PCS. If GMPCEN is read as logic 1, then the external PHY requires the internal PCS. If GMPCEN is read as logic 0, then the external PHY does not require the internal PCS.
28:10	RW	x	RESRVD	0	*Reserved.*
30:29	RW	x	GMRPP	0	*G/MII RPAT Pattern.*
31	RW	x	RPEN	0	*G/MII RPAT Enable.*

CSR 46 STATISTIC INDEX REGISTER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RESRVD								RESRVD								RESRVD								RESRVD					SCIX		

Bit Field	Type	E^2	Mnemonic	Default Value	Description
4:0	W	x	SCIX	0	*Statistic Index.* The index of the statistic whose value is to be placed into the *Statistic Value Register.* See Table 5-1 for a description of each statistic kept by the JT1001 controller.
31:5	x	x	RESRVD	0	*Reserved.*

This register is used in conjunction with the *Statistic Value Register* to read statistics maintained by the JT1001 controller. To read the values of a particular statistic, HOST software selects the statistic by writing the statistic index to this register. The act of selecting the index causes the JT1001 controller to take two actions: first, the current value of the statistic is latched into the *Statistic Value Register*, and second, the statistic is reset to 0. HOST software then reads the latched value of the selected statistic from the *Statistic Value Register.*

The table below defines the statistics kept by the JT1001 controller.

Table 5-1. Statistic Index Table

SCIX	Statistic Name	Description
0	aFramesTransmittedOK	Count of frames transmitted successfully by the JT1001 controller. This is an accurate count, despite the fact that the JT1001 controller performs lying sends. Frames that encounter single or multiple collisions are included in this count. Frames that encounter the maximum number of collisions are not included. The count does not wrap.
1	aSingleCollisionFrames	Count of frames that experienced single collision prior to successful transmission. The count does not include frames that encountered a late collision, multiple collisions, or the maximum number of collisions. The count does not wrap.
2	aMultipleCollisionFrames	Count of frames that experienced multiple collisions prior to successful transmission. The count does not include frames that encountered a late collision, a single collision, or the maximum number of collisions. The count does not wrap.
3	aFramesReceivedOK	Count of frames received without error and passed the JT1001 controller's destination address filter. If the VLAN support is enabled, the packet must also pass the VLAN filter. The count does not wrap.
4	aFrameCheckSequenceErrors	Count of frames received with FCS errors. This count does not include frames that had alignment errors. The count does not wrap.
5	aAlignmentErrors	A count of frames received with alignment errors. An alignment error occurs when the received frame is not an integral number of octets in length and does not pass the FCS check. The count does not wrap.

Table 5-1. Statistic Index Table (Continued)

SCIX	Statistic Name	Description
6	Dropped Packet Count	The number of dropped packets. A packet is considered dropped if it passed the JT1001 controller's destination address filter and VLAN filter, but could not be successfully received due to an internal error or lack of resource in the JT1001 controller. The count does not wrap.
7	Errored Receive Packet Count	The number of packets that encountered an error during packet reception. This register is the sum of all errors detected by the JT1001 controller during packet reception. The count does not wrap.
8	Errored Transmit Packet Count	The number of packets that encountered an error during transmission. This register is the sum of all errors detected by the JT1001 controller during packet transmission. This count does not include packets that encountered single or multiple collisions. The count does not wrap.
9	Late Collision Count	A count of frames that encountered a late collision during transmission. A late collision is defined as a collision that occurs after at least minimum frame size bytes of a frame has been transmitted. The count does not wrap.
10	Runt Packet Count	A count of frames received that were smaller than the minimum frame size of 64 bytes (including CRC). The count does not wrap.
11	aFrameTooLong	A count of frames received that were larger than the maximum packet size. The count does not wrap.
12	VLAN Accepted Packet Count	A count of VLAN tagged frames received that were accepted by the JT1001 controller. The count does not wrap.
13	VLAN Discarded Packet Count	A count of VLAN tagged frames received that were discarded by the JT1001 controller. The JT1001 controller discards VLAN tagged frames if the VLEN and VLTBEN bits are set in *Mode Register – 1* and the VLAN tag in the frame does not match any entries in the VLAN TCI Table. The count does not wrap.
14	TCP/IP IP Checksum Error Count	A count of TCP/IP packets that contained an IP header and failed the JT1001 controller's IP checksum test. This count is incremented regardless of the state of the PACKEREN bit in *Mode Register – 2*. The count does not wrap.
15	TCP/IP UDP Checksum Error Count	A count of TCP/IP packets that contained a UDP header and failed the JT1001 controller's UDP checksum test. This count is incremented regardless of the state of the PACKEREN bit in *Mode Register – 2*. The count does not wrap.
16	alnRangeLengthErrors	A count of packets that failed the JT1001 controller's packet length test. The count does not wrap.
17	TCP/IP TCP Checksum Error Count	A count of TCP/IP packets that contained a TCP header and failed the JT1001 controller's TCP checksum test. This count is incremented regardless of the state of the PACKEREN bit in *Mode Register – 2*. The count does not wrap.
18	TCP/IP Non Ipv4 Packet Count	A count of all packets transmitted that the JT1001 controller was requested to insert an IP, TCP, and/or UDP checksum, but could not do so because the version number in the IP header was not version 4. The count does not wrap.
19	aFramesAbortedDueToXSColls	A count of packets that experienced 16 collisions and failed to transmit. The count does not wrap.
20	Unicast Packets Received OK	A count of packets containing a unicast destination address that were received without error. The count does not wrap.
21	aMulticastFramesReceivedOK	A count of packets containing a multicast destination address that were received without error. The count does not wrap.
22	aBroadcastFramesReceivedOK	A count of packets containing a broadcast destination address that were received without error. The count does not wrap.
23	PAUSE Command Packets Received	A count of valid PAUSE packets received. The MAC Control Packets Received count is also incremented when this count increments. The count does not wrap.

Table 5-1. Statistic Index Table (Continued)

SCIX	Statistic Name	Description
24	PAUSE Command Packets Transmitted	A count of PAUSE command packets the JT1001 controller generated and transmitted. The count does not wrap.
25	MAC Control Packets Received	A count of MAC control packets received by the JT1001 controller. This count is independent of the PAUSE Command Packets Received count. The count does not wrap.
26	aFramesDeferredWithXmissions	A count of packets for which the first transmission was delayed because the network was busy. The count does not wrap.
27	aFramesWithExcessiveDeferral	A count of packets that were deferred greater than 3036-byte times before successful transmission. The count is incremented at most once per packet. The count does not wrap.
28	aCarrierSenseErrors	A count of times that carrier sense was deasserted during the transmission of a packet. The count is incremented at most once per packet. The count does not wrap.

CSR 47 STATISTIC VALUE REGISTER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SCVL																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	R	x	SCVL	0	*Statistic Value.* The value of the statistic selected by the last write to the *Statistic Index Register.*

CSR 48 VLAN TAG CONTROL INFORMATION TABLE

31–22	21	20	19–16	15–13	12	11–0
RESRVD	VLTBCM	RESRVD	VLTBIX	VLUSPR	RESRVD	VLID

Bit Field	Type	E^2	Mnemonic	Default Value	Description
11:0	W	x	VLID	0	*VLAN Identifier.* This field corresponds to the VLAN Identifier field in the VLAN tag header. When receiving a VLAN tagged packet, the JT1001 controller uses this field to determine if the packet will be accepted or rejected. When the JT1001 controller is inserting a VLAN tag header prior to packet transmission, this field is used in the construction of the VLAN tag header.
12	x	x	RESRVD	0	*Reserved.*
15:13	W	x	VLUSPR	0	*VLAN User Priority.* The field corresponds to the user_priority field in the VLAN tag header. When receiving a VLAN tagged packet, the JT1001 controller uses this field to determine if the packet will be accepted or rejected. When the JT1001 controller is inserting a VLAN tag header prior to packet transmission, this field is used in the construction of the VLAN tag header.
19:16	W	x	VLTBIX	0	*VLAN TCI Table Index.* The index of the VLAN TCI Table entry to be acted upon by the command specified by VLTBCM. The TCI Table has 16 entries. The entry at index 0 is the “global” entry.
20	x	x	RESRVD	0	Reserved.
21	W	x	VLTBCM	0	*VLAN TCI Table Command.* This bit indicates the operation to be performed on the VLAN TCI Tag Table. When set, the VLID and VLUSPR information is added to the table at the index specified by VLTBIX. When the VLTBCM bit is clear, the TCI information at index VLTBIX in the VLAN TCI Tag Table is deleted.
31:22	x	x	RESRVD	0	*Reserved.*

CSR 49 VLAN TAG PROTOCOL ID REGISTER

31–16	15–0
RESRVD	TPID

Bit Field	Type	E^2	Mnemonic	Default Value	Description
15:0	W	x	TPID	8888h	*VLAN Tag Protocol Identifier.* This value is the tag protocol identifier (TPID) of the VLAN tag header. The JT1001 controller uses this value when constructing a VLAN tag header to be inserted into the packet to be transmitted. When the VLEN bit is set in *Mode Register – 1,* the JT1001 controller uses this value when filtering received packets that contain a VLAN tag.
31:16	x	x	RESRVD	0	*Reserved.*

CSR 50 RESERVED

31–0
RESRVD

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	R	x	RESRVD	0	*Reserved.*

CSR 51 COMMAND STATUS REGISTER

31 30	29 28 27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12 11 10	09 08 07 06 05 04 03 02 01 00
RESRVD	RXCMFECN	RESRVD	TXCMFECN	RXDMDNCN	TXDMDNCN

Bit Field	Type	E^2	Mnemonic	Default Value	Description
7:0	RC	x	TXDMDNCN	0	*Transmit DMA Done Count.* Indicates how many TX PDL/PDCs the JT1001 controller has completed processing since the last read of this field. HOST software uses this count to determine when TX PDL/PDC buffers are no longer in use by the JT1001 controller. The JT1001 controller resets the count after each read of this field. NOTE: This field can be accessed as a BYTE, WORD, or DWORD. A BYTE read of this field has no affect on other fields in the CSR. More specifically, the RXDMDNCN count is not changed by a BYTE access to this field.
15:8	RC	x	RXDMDNCN	0	*Receive DMA Done Count.* The count of RX PDL/PDCs the JT1001 controller has completed processing since the last read of this field. HOST software uses this field to determine how many RX PDL/PDC buffers have been filled with receive data by the JT1001 controller. The JT1001 controller resets the count to 0 after each read. NOTE: This field can be accessed as a BYTE, WORD, or DWORD. A BYTE read of this field has no affect on other fields in the CSR. More specifically, the TXDMDNCN count is not changed by a BYTE access to this field. NOTE: This field is aliased into the *Event Status Register.* Whether the count is read via the *Command Status Register* or the *Event Status Register,* the JT1001 controller will reset the count after the read.
21:16	R	x	TXCMFECN	31	*Transmit Command Free Count.* This is the number of transmit command queue entries free in the chip. HOST software uses this field to determine how many additional transmit commands can be queued to the chip. The JT1001 controller can queue a maximum of 31 transmit commands.
23:22	x	x	RESRVD	0	*Reserved.*
29:24	R	x	RXCMFECN	31	*Receive Command Free Count.* This is the number of receive command queue entries free in the chip. HOST software uses this field to determine how many additional receive commands can be queued to the chip. The JT1001 controller can queue a maximum of 31 receive commands.
31:30	x	x	RESRVD	0	*Reserved.*

CSR 52 FLOW CONTROL WATERMARK REGISTER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FLCTHIWM																FLCTLOWM															

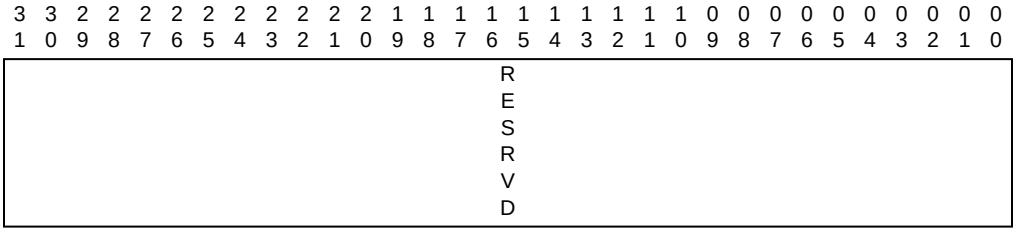
Bit Field	Type	E^2	Mnemonic	Default Value	Description
15:0	RW	x	FLCTLOWM	500h	*Flow Control Low Watermark.* This field defines the flow control low watermark. The watermark is expressed in terms of the number of DWORDs in use in the RX FIFO. When the RX FIFO reaches the high flow control high watermark and then falls to the low watermark, the JT1001 controller constructs and transmits a PAUSE frame with the pause duration set to 0. This PAUSE frame informs the link partner that the congested condition has subsided and it may begin transmitting immediately.
31:16	RW	x	FLCTHIWM	3B00h	*Flow Control High Watermark.* This field defines the flow control high watermark. The watermark is expressed in terms of the number of DWORDs in use in the RX FIFO. When the RX FIFO reaches the high watermark, the JT1001 controller constructs and transmits a PAUSE frame instructing the link partner to stop transmitting for 0FFFFh*512 bit times. As long as the RX FIFO stays above the low watermark, the JT1001 controller sends additional PAUSE frames at an interval slightly less 0FFFFh*512 bit times. This has the effect of keeping the link partner paused.

CSR 53 RESERVED

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESRVD																															

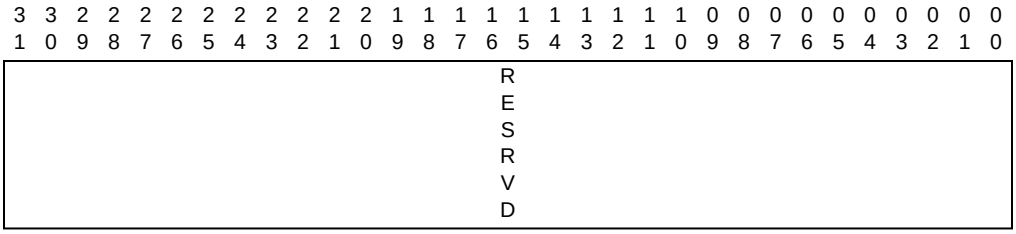
Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	x	x	RESRVD	0	*Reserved.*

CSR 54 RESERVED



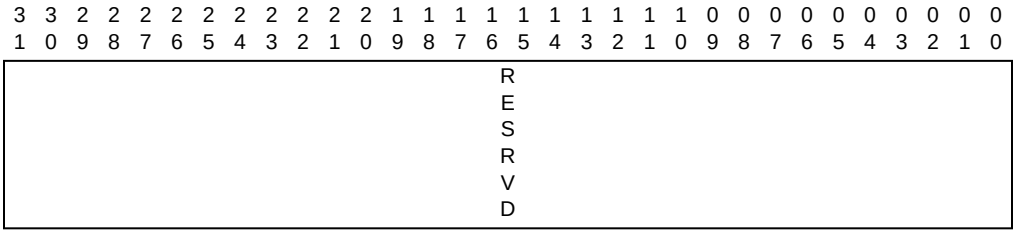
Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	x	x	RESRVD	0	*Reserved.*

CSR 55 RESERVED



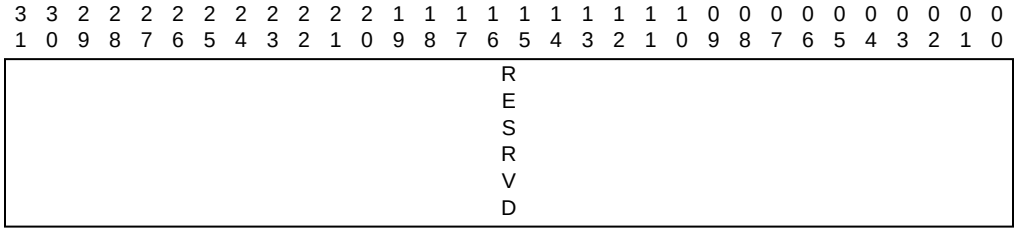
Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	x	x	RESRVD	0	*Reserved.*

CSR 56 RESERVED



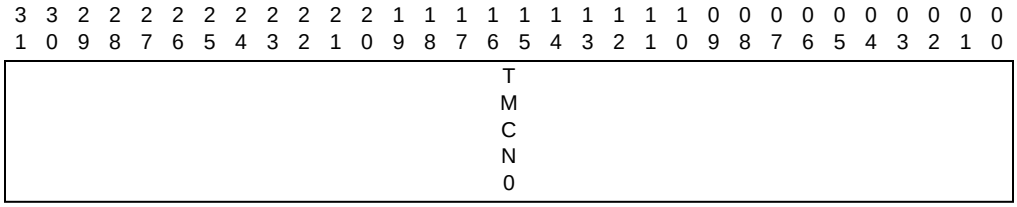
Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	x	x	RESRVD	0	*Reserved.*

CSR 57 RESERVED



Bit Field	Type	E²	Mnemonic	Default Value	Description
31:0	x	x	RESRVD	0	*Reserved.*

CSR 58 TIMER 0 COUNT REGISTER



Bit Field	Type	E²	Mnemonic	Default Value	Description
31:031:0	RW	x	TMCN0	0	NOTE: The definition of this register is temporary and will be changed in a future revision of the JT1001 controller. *Timer 0 Count.* The timer tick count. This count increments with each timer tick when the TMEN0 bit is set in the *Command Register.* The timer operates at 25 MHz and therefore ticks once every 40 ns. The count remains constant when the TMEN0 bit is clear. The count does not wrap. A write to this register causes the counter to be reset to 0, regardless of the value actually written.

CSR 59 TIMER 0 INTERRUPT TRIGGER REGISTER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
TMINTR0																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	RW	x	TMINTR0	FFFFFFFFh	NOTE: The definition of this register is temporary and will be changed in a future revision of the JT1001 controller. *Timer 0 Interrupt Trigger.* This register specifies an interrupt threshold for Timer 0. If the *Timer 0 Count Register* equals the value in this register, and the TMMS0 bit in the *Interrupt Mask Register* is set, the JT1001 controller will generate an interrupt.

CSR 60 TIMER 1 COUNT REGISTER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
TMCN1																															

Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	RW	x	TMCN1	0	NOTE: The definition of this register is temporary and will be changed in a future revision of the JT1001 controller. *Timer 1 Count.* The timer tick count. The count increments with each timer tick when the TMEN1 bit is set in the *Command Register.* The timer operates at 25 MHz and, therefore, ticks once every 40 ns. The count remains constant when the TMEN1 bit is clear. The count does not wrap. A write to this register causes the counter to be reset to 0, regardless of the value actually written.

CSR 61 TIMER 1 INTERRUPT TRIGGER REGISTER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
TMINTR1																															

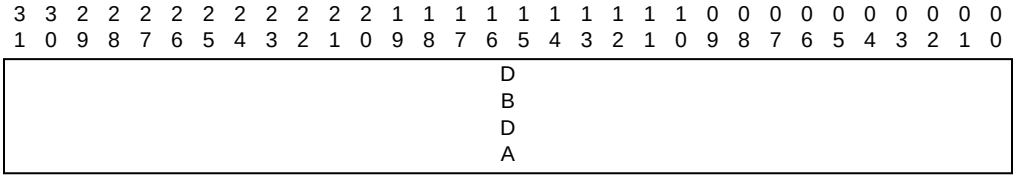
Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	RW	x	TMINTR1	FFFFFFFFh	NOTE: The definition of this register is temporary and will be changed in a future revision of the JT1001 controller. *Timer 1 Interrupt Trigger.* This register specifies an interrupt threshold for Timer 1. If the *Timer 1 Count Register* equals the value in this register, and the TMMS1 bit in the *Interrupt Mask Register* is set, the JT1001 controller will generate an interrupt.

CSR 62 DEBUG COMMAND REGISTER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
DBST	RESRVD																														BIST

Bit Field	Type	E^2	Mnemonic	Default Value	Description
0	W	x	BIST	0	*Built In Self Test.* This command causes the JT1001 controller to perform a built-in self test (BIST). After issuing this command, HOST software polls the DBST until it clears. When the JT1001 controller completes the BIST, it places the BIST completion code into the *Debug Data Register* and clears DBST. HOST software can then read the *Debug Data Register* to obtain the completion code for BIST. A 0 completion code indicates the BIST passed. A non-zero completion code indicates BIST failed.
30:1	x	x	RESRVD	0	*Reserved.*
31	RA	x	DBST	0	*Debug Status Bit.* This bit is set by HOST software when writing to the *Debug Address Register.* When the requested action is complete, the JT1001 controller clears the bit. HOST software polls this bit to determine when the command has completed.

CSR 63 DEBUG DATA REGISTER



Bit Field	Type	E^2	Mnemonic	Default Value	Description
31:0	RW	x	DBDA	0	*Debug Data Register.* The value and interpretation of this field is dependent on the command issued in the *Debug Command Register.* Depending on the command issued, HOST software may need to write this register before issuing the command, or read this register after issuing the command.

Section 6
Register Placement

Register	Description	Location
CSR00	Mode Register – 1	SIB
CSR01	Mode Register – 2	SIB
CSR02	Transmit PDC Buffer Address Table Index	SIB
CSR03	Product Identification Register	
CSR04	Transmit PDC Buffer Address LSD	SIB
CSR05	Transmit PDC Buffer Address MSD	SIB
CSR06	Receive PDC Buffer Address Table Index	SIB
CSR07	Reserved	
CSR08	Receive PDC Buffer Address LSD	SIB
CSR09	Receive PDC Buffer Address MSD	SIB
CSR10	EEPROM	SIB
CSR11	Chip Status Register	SIB
CSR12	TX PDL Address Register LSD	SIB
CSR13	TX PDL Address Register MSD	SIB
CSR14	RX PDL Address Register LSD	SIB
CSR15	RX PDL Address Register MSD	SIB
CSR16	TX PDC Register	SIB
CSR17	RX PDC Register	SIB
CSR18	Interrupt Period Register	SIB
CSR19	TX FIFO Packet Count Register	DBS
CSR20	TX FIFO Low Watermark Register	DBS
CSR21	TX FIFO DWORDs Free Register	DBS
CSR22	TX FIFO Write Register	DBS
CSR23	Reserved	
CSR24	RX FIFO Read Register	DBS
CSR25	Reserved	
CSR26	RX FIFO DWORD Count Register	DBS
CSR27	RX FIFO Watermark Register	DBS
CSR28	RX FIFO Packet Count Register	DBS
CSR29	Command Register	DBS
CSR30	Interrupt Mask Register	SIF
CSR31	Reserved	
CSR32	Event Status Register	SIF
CSR33	Reserved	

Register	Description	Location
CSR34	Multicast Hash Table Register LSD	Just outside of the MAC
CSR35	Multicast Hash Table Register MSD	Just outside of the MAC
CSR36	LED 0 Configuration Register	SIF
CSR 37	LED 1 Configuration Register	SIF
CSR38	LED 2 Configuration Register	SIF
CSR39	LED 3 Configuration Register	SIF
CSR40	Reserved	
CSR41	EEPROM Data Register	SIF
CSR42	LAN Physical Address Register LSD	Just outside of the MAC
CSR43	LAN Physical Address Register MSW	Just outside of the MAC
CSR44	G/MII PHY Access Register	PHY
CSR45	G/MII Mode Register	PHY
CSR46	Statistic Index Register	SIF
CSR47	Statistic Value Register	SIF
CSR48	VLAN TCI Table Register	SIF
CSR49	VLAN Tag Protocol ID Register	SIF
CSR50	Reserved	
CSR51	Command Status Register	SIF
CSR52	Flow Control Watermark Register	
CSR53	Reserved	
CSR54	Reserved	
CSR55	Reserved	
CSR56	Reserved	
CSR57	Reserved	
CSR58	Timer 0 Count Register	
CSR59	Timer 0 Interrupt Trigger Register	
CSR60	Timer 1 Count Register	
CSR61	Timer 1 Interrupt Trigger Register	
CSR62	Debug Command Register	
CSR63	Debug Data Register	

Section 7
EEPROM Map

Off	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00	REGISTER
00	Reserved																																Unused
01	SUBSYSTEM ID																SUBSYTEM VENDOR ID																PCI CFG
02	MAX LATENCY								MIN GRANT								RESERVED																PCI CFG
03	LNCKEN	USPIMD1	USPIMD0	VLISGB	VLRMID	VLTBEN	VLEN	RESERVED	RXFLCTEN	MGMCBEN	MGPKEN	DBMDEN	RESERVED				LGPKEN	RESERVED							RMPPEN	TXPPEN	GMSTPOEN	RXTRPR	TXFLCTEN	RESERVED				CSR 00
04	RESERVED															CA	CO	JA	RX	TX	ADMA	LKST	AN	FD	RESERVED	1000MB	100MB	10MB	PUXP	LDIPSGPL	LDEN	RESERVED	CSR 36
05	RESERVED															CA	CO	JA	RX	TX	ADMA	LKST	AN	FD	RESERVED	1000MB	100MB	10MB	PUXP	LDIPSGPL	LDEN	RESERVED	CSR 37
06	RESERVED															CA	CO	JA	RX	TX	ADMA	LKST	AN	FD	RESERVED	1000MB	100MB	10MB	PUXP	LDIPSGPL	LDEN	RESERVED	CSR 38
07	RESERVED															CA	CO	JA	RX	TX	ADMA	LKST	AN	FD	RESERVED	1000MB	100MB	10MB	PUXP	LDIPSGPL	LDEN	RESERVED	CSR 39
08	RESERVED																PHAD5								PHAD4								CSR 43
09	PHAD3								PHAD2								PHAD1								PHAD0								CSR 42

Figure 7-1. EEPROM Map

Off	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00	REGISTER
10	RESERVED										FLWTEN	FLPN	EXRMTM				RESERVED																CSR 10
11	RESERVED																TPID																CSR 49
12	RESERVED																PACKEREN	RXUPCKEN	RXTPCKEN	RXIPCKEN	TXUPCKEN	TXTPCKEN	TXIPCKEN	RESERVED										CSR 01
• • •	RESERVED																																UNUSED
31	CHECKSUM																																CHECKSUM

Figure 7-1. EEPROM Map (Continued)

Section 8
Glossary

Symbol	Description
AB	Arbitrate
AC	Acceptable
AD	Address
AL	Alignment
AN	Auto Negotiation
AV	Available
BC	Broadcast
BF	Buffer
BK	Back
BT	Byte
BU	Bus
CA	Carrier
CD	Code
CK	Check
CL	Control
CM	Command
CN	Count
CO	Collision
CR	Cyclic Redundancy Check (CRC)
CS	Chip Select
CT	Control
DA	Data
DB	Debug
DE	Descriptor
DF	Defined
DL	Delayed
DM	Direct Memory Access. Refers to transactions initiated by the JT1001 controller on the HOST/JT1001 controller interconnect bus portion directly attached to the JT1001 controller. For JT1001, DMA refers to MASTER PCI cycles initiated by the JT1001.

Symbol	Description
DN	Done
DR	Dropped
DS	Disable
DV	Device
DW	Double Words (4 bytes)
E^2	Electrically Erasable (as in EEPROM)
EM	Empty
EN	Enable
ER	Error
EX	Exhausted/Expired
FA	Failure
FD	Full Duplex
FE	Free
FI	FIFO
FL	Flow
FR	Frame
GM	GMII
GN	General
HD	Header
HI	High
HS	Hash
HT	Hit
HW	Hardware
ID	Identifier
IL	Idle
IN	Interrupt
IP	Input or Internet Protocol (IP)
IS	Insert
IX	Index
JA	Jabber
LA	Layer
LD	LED
LG	Long
LK	Link
LN	Line
LO	Low
LP	Loop
LT	Late

Symbol	Description
MA	Match
MB	Megabit
MC	Multicast
MD	Mode
MG	Magic
MI	MII
MS	Mask
MU	Multiple
NV	Invalidate
OD	Order
OK	Okay
OU	Out
OV	Over
PA	Pause
PE	Periodic
PD	PDL or PDC
PH	Physical
PI	Pin
PK	Packet
PL	Polarity
PM	PROM
PN	Present
PO	Promiscuous or Poll
PP	Packet Pad
PR	Priority
PS	Pass
PT	Pointer
PU	Pulse
RD	Read
RE	Reset
RESRVD	Reserved
RG	Ring
RM	ROM or Remove
RQ	Request
RR	Register
RT	Retries
RU	Runt
RV	Revision

Symbol	Description
RX	Receive
SC	Statistic
SE	Set
SG	Signal
SI	Single
SK	Skip
SL	Slave
SM	Sum
SP	Speed
SR	Stretcher
ST	State/Status
SW	Software
SY	Symbol
TB	Table
TM	Timer
TP	Transmission Control Protocol (TCP)
TR	Trigger
TS	Test
TX	Transmit
UC	Unicast
UP	User Datagram Protocol (UDP)
US	User
VL	Value or VLAN
WM	Watermark
WR	Wire
WT	Write
XP	Expander

JATO TECHNOLOGIES, INC.
505 E. HUNTLAND DRIVE, SUITE 550
AUSTIN, TX 78752

(512) 407-2100

http://www.jatotech.com