
# IXP1200 Microcode/ StrongARM Core Interface Control Document (ICD)

## Version 1.2

www.radisys.com

World Headquarters
5445 NE Dawson Creek Drive • Hillsboro, OR
97124 USA
Phone: 503-615-1100 • Fax: 503-615-1121
Toll-Free: 800-950-0044

International Headquarters
Gebouw Flevopoort • Televisieweg 1A
NL-1322 AC • Almere, The Netherlands
Phone: 31 36 5365595 • Fax: 31 36 5365620

RadiSys Microware Communications Software Division, Inc.
1500 N.W. 118th Street
Des Moines, Iowa 50325
515-223-8000

Revision A
December 2001

## Copyright and publication information

This manual reflects version 1.2 of IXP1200 Microcode Solutions Library.
Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Microware Communications Software Division, Inc.

## Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, RadiSys Corporation will not be liable for any damages including indirect or consequential, from use of the OS-9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

## Reproduction notice

The software described in this document is intended to be used on a single computer system. RadiSys Corporation expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

December 2001

# Table of Contents

# Chapter 1: IXP1200 Microcode/StrongARM Core Interface Control

This chapter describes the interface presented by the IXP1200 microcode to the StrongARM core. It includes the following sections:

- **Introduction**
- **Buffer Descriptors**
- **CSR Description**
- **Functional Description**
- **Understanding Queue Functionality**

# Introduction

The interface between the IXP1200 microcode and the StrongARM core, as described in this document, provides the basic framework necessary to write an Ethernet and/or ATM driver for any operating system running on the StrongARM core.

The StrongARM core driver has the following initialization responsibilities:

- To ensure that the hardware has been correctly initialized.
- To provide the required configuration parameters to the microengines.

The configuration information is provided to the microengines via a configuration and status register (CSR) block.

After initialization of the interfaces, the StrongARM core is responsible only for management of the high level buffer descriptors. All MAC and physical layer processing is handled by the microengines.

## For More Information

**Appendix A: Data Structures and Layout** contains a 'C' language structure that describes the CSR interfaces.

# Buffer Descriptors

The core processor and microengines use buffer descriptors to exchange the location of transmitted and received data as well as any required control information.

In the current implementation the Ethernet and ATM drivers share a common pool of buffer descriptors. The buffer descriptor pool is divided into separate transmit buffer descriptor and receive buffer descriptor pools. Thus there are two distinct groups.

- ATM/Ethernet Receive
- ATM/Ethernet Transmit

## Data Structure Format

While the buffer descriptor pools are shared between ATM and Ethernet, the status bits are different depending on which interface type is being used. The format of the buffer descriptors is shown in **Figure 1-1**.

**Figure 1-1 Buffer Descriptor Data Structure Format**

| | | |
|---|---|---|
| Offset 0 | Next Buffer Descriptor | |
| Offset 4 | Status | Port Number |
| Offset 8 | Data Length | Reserved |
| Offset 12 | Data Size | Reserved |
| Offset 16 | Data Location | |
| Offset 20 | User Data | |
| Offset 24 | Multiport | |
| Offset 28 | Reserved | |

Each buffer descriptor refers to a single complete packet. At a minimum there must be at least one transmit and one receive buffer descriptor for each driver although in any practical application more are required. The number required depends on how many ports will be active and the

expected packet rates. Performance will be impacted if either the StrongARM core driver or microengines are unable to get a free buffer descriptor. The maximum number of descriptors is limited only by the amount of available memory. It is strongly recommended that all buffer descriptors either be located in non cached memory, or no more than one occupy a single cache line.

### Linked List Queues

The memory location of the various buffer descriptor queues is configured via the CSR. There are two basic types of linked list queues. The first type makes use of the IXP push/pull stacks. Three of the available eight push/pull stacks are required for either of the ATM/Ethernet interfaces. These queues hold the transmit and receive free lists in addition to a list of transmitted buffer descriptors awaiting final StrongARM core processing. No locking is required when inserting onto or removing from these queues. The hardware provides this feature.

The second type of queue is used for the transmit and receive stacks. These must maintain a FIFO ordering, therefore the push/pull stacks are not appropriate. To provide fast insertions and deletions, a head and tail pointer are maintained for each of these queues. In order to maintain consistency, a locking mechanism must be used, such as CAM locking or semaphore locking.

## Receive Buffer Descriptors

The StrongARM core driver is responsible for allocating and initializing receive buffer descriptors. These buffer descriptors should be placed on the `rxfree` stack before enabling the hardware receiver.

## Control/Status (Ethernet)

The ATM and Ethernet drivers share a common pool of buffer descriptors, however the control/status information field has some bits defined differently. **Figure 1-2** shows the format of the control/status field for Ethernet.

**Figure 1-2 Control/Status (Ethernet)**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | | | | | T | | | FC | FR | OV | CR | SH | LG | MC | BC |

The first 16 bits of a receive buffer descriptor contain the control/status information. All reserved fields should be initialized to zero.

| | |
|---|---|
| Bit 15 | Interrupt<br>Setting this bit causes the microengines to generate an interrupt to the StrongARM core after filling this buffer descriptor with data and placing it on the `rxbd` queue. Currently this bit must be set in all receive buffer descriptors. |
| Bits 14 through 11 | Reserved |
| Bit 10 | Type<br>This bit indicates whether this is a receive or transmit buffer descriptor. For receive buffer descriptors the StrongARM driver should set it to 0. |
| Bits 9 through 8 | Reserved |
| Bit 7 | Flow control packet<br>Indicates the received packet is a flow control packet. |
| Bit 6 | Framing error<br>Indicates the packet was received with a framing error. |

| | |
|---|---|
| Bit 5 | Receiver overrun<br>Indicates the microcode was unable to remove data from the hardware receive FIFO before it overflowed. This indicates either the microcode is not fast enough to keep up, or the system ran out of free receive buffer descriptors. |
| Bit 4 | CRC error<br>Indicates the packet was received correctly, but the CRC was incorrect. |
| Bit 3 | Short packet<br>Indicates a packet smaller than 64 bytes was received. |
| Bit 2 | Long packet<br>Indicates a packet longer than the maximum was received. |
| Bit 1 | Multicast<br>Indicates the packet was received as a hardware layer multicast. |
| Bit 0 | Broadcast<br>Indicates the packet was received as a hardware layer broadcast. |

## Control/Status (ATM)

The ATM and Ethernet drivers share a common pool of buffer descriptors, however the control/status information field has some bits defined differently. **Figure 1-3** shows the format of the control/status field for ATM.

**Figure 1-3 Control/Status (ATM)**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | | | | | T | | | | | | | | | | |

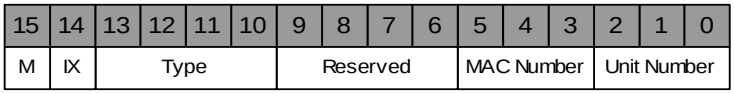
The first 16 bits of a receive buffer descriptor contain the control/status information.

| | |
|---|---|
| Bit 15 | Interrupt<br>Setting this bit causes the microengines to generate an interrupt to the StrongARM core after filling this buffer descriptor with data and placing it on the `rxbd` queue. Currently this bit must be set in all receive buffer descriptors. |
| Bits 14 through 11 | Reserved |
| Bit 10 | Type<br>This bit indicates whether this is a receive or transmit buffer descriptor. For receive buffer descriptors this should be set to 0. |
| Bits 9 through 0 | Reserved |

## Port Number

For receive buffer descriptors the port number field will be set by the microcode to indicate on which interface the packet was received. **Figure 1-4** shows the port number field.

**Figure 1-4 Port Number Field**



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| M | IX | Type | | | | Reserved | | | | MAC Number | | | Unit Number | | |

Bit 15 — Multiport

If this bit is set it indicates the multiport field contains a valid port bitmap. On a receive buffer descriptor this is only used internally by the microcode.

Bit 14 — IX bus

This bit indicates the port is located on the IX bus. For both Ethernet and ATM this will be set to 1 by the microcode.

Bits 13 through 10 — Type of interface

1 - Ethernet

2 - ATM

3 - Switch Fabric

4 through 15: Reserved

Bits 9 through 6 — Reserved

Bits 5 through 3 — MAC number

Bits 2 through 0 — Unit number

## Data Length

The data length must be initialized to 0 before being placed on the `rxfree` stack. Before the microengines enqueue the buffer descriptor on the `rxbd` queue, the data length field will be set to reflect the total number of bytes copied into the buffer. This includes both the data link header and payload. In the case of ATM, only the ATM header on the first cell is copied, and subsequent ATM headers for the same packet are discarded.

## Data Size

This field indicates the maximum number of data bytes the microengines can copy into the buffer associated with this buffer descriptor. For both Ethernet and ATM it must be large enough to contain an entire frame. This means it should be at least 1552 for ATM and 1514 for Ethernet.

## Data Location

The data location is a pointer to the beginning of the buffer associated with this buffer descriptor. For receive buffer descriptors this must be on an 8 byte boundary. The ATM microcode also requires the 8 bytes immediately preceding this location to be available to store temporary data.

## User Data

This field can be used to store any data needed by the driver. This field is never looked at or modified by the microengines. For example, if the data location pointer points inside of a larger data structure this field could be used to find the encompassing data structure.

## Multiport

In receive buffer descriptors this field is unused by the StrongARM driver and should be initialized to 0.

### Next Buffer Pointer

Used to link buffer descriptors together on the various queues. This should always be set to NULL by the StrongARM driver before enqueuing the buffer descriptor on any queue.

## Transmit Buffer Descriptors

When the core transmits a packet, the driver removes a buffer descriptor from the `txfree` stack. It then fills in all the required transmit fields and enqueues it on the `txbd` queue. The core driver does not need to wait for the packet to be sent, but instead can immediately use the next transmit buffer descriptor. This may continue until either the core processor runs out of packets to transmit, or the `txfree` list is empty.

Once the core processor enqueues the packet it is not allowed to change any fields in the buffer descriptor. After each packet has been sent the microengines enqueue the buffer descriptor on the `txdone` stack and interrupt the StrongARM core if requested. The StrongARM driver then removes buffer descriptors from this stack, frees any associated system memory, and places the buffer descriptor back in the `txfree` stack.

### Control/Status (Ethernet)

The ATM and Ethernet drivers share a common pool of buffer descriptors, however the control/status information field has some bits defined differently. **Figure 1-5** shows the format of the control/status field for Ethernet.

**Figure 1-5 Control/Status (Ethernet)**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | | | | | T | | | | | | | | | | |

The first 16 bits of a transmit buffer descriptor contain the control/status information.

| | |
|---|---|
| Bit 15 | Interrupt<br>Setting this bit will cause the microengines to send an interrupt to the StrongARM core after transmitting the data and placing the buffer descriptor on the `txdone` stack. Currently this bit must be set in all transmit buffer descriptors. |
| Bits 14 through 11 | Reserved |
| Bit 10 | Type<br>This bit indicates whether this is a receive or transmit buffer descriptor. For transmit buffer descriptors this must be set to 1. |
| Bits 9 through 0 | Reserved |

## Control/Status (ATM)

The ATM and Ethernet drivers share a common pool of buffer descriptors, however the control/status information field has some bits defined differently. **Figure 1-6** shows the format of the control/status field for ATM.

**Figure 1-6 Control/Status (ATM)**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | | | | | T | | | | | | | | | | |

The first 16 bits of a transmit buffer descriptor contain the control/status information.

| | |
|---|---|
| Bit 15 | Interrupt<br>Setting this bit causes the microengines to send an interrupt to the StrongARM core after transmitting the data and placing the buffer descriptor on the txdone stack. Currently this bit must be set in all transmit buffer descriptors. |
| Bits 14 to 11 | Reserved |
| Bit 10 | Type<br>This bit indicates whether this is a receive or transmit buffer descriptor. For transmit buffer descriptors this must be set to 1. |
| Bits 9 to 0 | Reserved |

## Port Number

For transmit buffer descriptors the port number field is used to indicate on which port(s) the packet should be transmitted. **Figure 1-7** shows the port number field.

**Figure 1-7 Port Number Field**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| M | IX | Type | | | | Reserved | | | | MAC Number | | | Unit Number | | |

| | |
|---|---|
| Bit 15 | Multiport<br>If this bit is set it indicates the multiport field contains a valid port bitmap. Also, if this bit is set the remaining port number fields are undefined and should be ignored. |

| | |
|---|---|
| Bit 14 | IX bus<br>This bit indicates the port is located on the IX bus. For Ethernet and ATM this bit should be set to 1 before transmitting a packet. |
| Bits 13 through 10 | Type of interface<br>1 - Ethernet<br>2 - ATM<br>3 - Switch Fabric<br>4 through 15: Reserved |
| Bits 9 through 6 | Reserved |
| Bits 5 through 3 | MAC number |
| Bits 2 through 0 | Unit number |

## Data Length

This field indicates the number of data bytes in the buffer associated with this descriptor that should be transmitted.

## Data Size

The data size fields are not used by transmit buffer descriptors, and should be initialized to zero.

## Data Location

The data location field points to the beginning of a buffer containing the packet to be transmitted. The packet must already contain the Ethernet or ATM data link header. In the case of ATM only the header for the first cell is included.

There are no alignment restrictions on the data location pointer for Ethernet. For ATM the data location must be 8 byte aligned.

## User Data

This field can be used to store any data needed by the driver. This field is never looked at or modified by the microengines. For example, if the data location pointer points inside of a larger data structure this field could be used to find the encompassing data structure.

## Multiport

This field is used when sending a packet to multiple ports. Each of the 32 bits represents a different port. Bit 0 is MAC 0 Unit 0, bit 1 is MAC 0 unit 1, up to bit 31 which is MAC 3 unit 7. The bitmap contained in this field is only valid if the multiport bit is set in the port number field.

## Next Buffer Pointer

Used to link buffer descriptors together on the various queues. This should always be set to NULL by the StrongARM driver before enqueuing the buffer descriptor on any queue.

# CSR Description

The configuration and status registers (CSR) for the Ethernet and ATM interfaces are defined in **Appendix A: Data Structures and Layout**. The CSRs for Ethernet and ATM are divided into two sections and contain configurable parameters to control microengine operation. The first is a single common section that is shared by all interfaces, and a second section that contains port specific information.

## Common CSR

The common CSR section contains all the information shared by the Ethernet and ATM ports.

### Queuing Data Structures

| | |
|---|---|
| `rxfree_push` | A pointer to one of the eight IXP push stacks. When the StrongARM core is finished processing a received buffer descriptor, the driver should return it to the `rxfree` list using this push stack. |
| `rxfree_pull` | A pointer to one of the eight IXP pull stacks. Available to microengines that require a receive buffer descriptor. This parameter must be associated with the same push/pull stack as `rxfree_push`. |
| `txfree_push` | Similar to `rxfree_push`. The StrongARM core driver should enqueue transmit buffer descriptors here after processing them from the `txdone_pull` stack. |
| `txfree_pull` | Similar to `rxfree_pull`. When the StrongARM core driver needs a free transmit buffer descriptor it can get one from this stack. |

| | |
|---|---|
| `txdone_push` | A pointer to an IXP push stack that stores transmitted buffer descriptors. After the microengines are finished with a transmit buffer descriptor it will be enqueued here. |
| `txdone_pull` | The StrongARM core driver can retrieve transmitted buffer descriptors from this IXP pull stack. This allows the driver to free any memory associated with a buffer descriptor before it is enqueued on the `txfree_push` stack. |
| `rxbd_tail` | A pointer to the tail of the receive stack. The microengines will enqueue any incoming packets here. |
| `rxbd_head` | A pointer to the head of the receive stack of buffer descriptors. The StrongARM core driver will receive incoming packets from here. |
| `txbd_tail` | A pointer to the tail of the transmit stack. The StrongARM core driver should enqueue all transmit buffer descriptors here. |
| `txbd_head` | A pointer to the head of the transmit stack. The microengines will remove buffer descriptors that are ready to be transmitted from here. |

## Interrupt Vector (vector)

This contains the base vector used by the microcode. Each microengine thread that potentially may generate an interrupt to the StrongARM core uses a unique vector. That vector number is determined by adding the thread number to the base vector contained in this register.

## Interrupt Event/Mask Register (i_event/i_mask)

The event register and mask register are physically different registers but contain the same bit field definitions. The event register indicates which of the various interrupts are pending. However, an actual interrupt to the StrongARM core only occurs if the equivalent bit in the mask register is set. **Figure 1-8** shows the format of the interrupt event/mask register.

**Figure 1-8 Interrupt Event/Mask Register**

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

Bit 31 — Transmit Buffer

This interrupt is generated by the microengines after a buffer has been transmitted and the associated buffer descriptor placed on the `txdone` stack.

Bit 30 through 15 — Reserved

Bit 14 — Receive Error

If an error occurs during the reception of a packet the receive error interrupt is generated. The buffer descriptor is still enqueued on the `rxbd` queue and the `bd_status` field indicates the cause of the error.

Bit 13 — Receive Frame

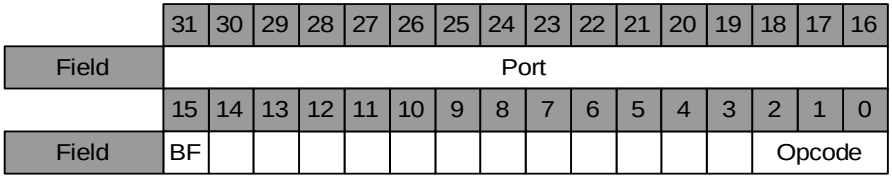
After a frame has been successfully received, it is placed in the `rxbd` queue and the receive frame interrupt is generated.

Bit 1 through 0 — Reserved

## Microengine Command Register

The microengine command register (mccr) sends commands to the microengine. **Figure 1-9** shows the format of the mccr.

**Figure 1-9 Microengine Command Register**




Bits 31 through 16 — Port

The port field is a bit field specifying which Ethernet ports on which to apply the command. Bit 16 represents port 0 and bit 31 represents port 15. It is possible for a single command to be applied to multiple ports by setting more than one bit in this field.

Bit 15 — Busy Flag

This bit indicates the microengines are currently executing a command. The driver sets this bit after initializing the appropriate parameters and then waits for the bit to be cleared, indicating the command has finished.

Bits 3 through 0 — Opcode

The opcode field specifies what command is to be executed on the indicated port(s).

0000: Start transmitter and receiver

It is possible to start both the transmitter and receiver for a port at the same time. This is equivalent to a start receiver followed by a start transmitter command.

0001: Start receiver

0010: Start transmitter

0011: Stop receiver

0100: Stop transmitter

# Port Specific CSR

Each ATM and Ethernet port has its own copy of the port specific CSR. Even though all of the fields are in each of the port specific entries, some of them apply only to ATM or only to Ethernet. The value of the unused fields is undefined and should not be used.

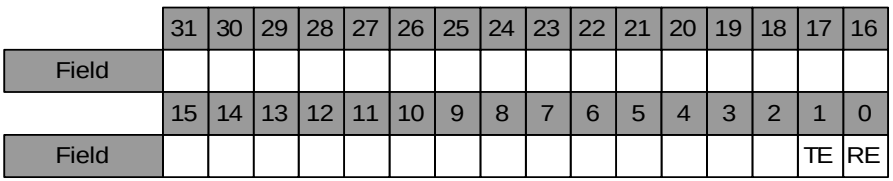
## Physical Address (paddr) - Ethernet Only

Each Ethernet port must be configured with a 48 bit 802.3 MAC layer address.

## Transmit/Receive Control Register

The transmit/receive control register (trcr) configures the transmitter and receiver. **Figure 1-10** shows the format of the (trcr).

**Figure 1-10 Transmit/Receive Control Register**




Bits 31 through 2 Reserved

| | |
|---|---|
| Bit 1 | Transmit enable |
| Bit 0 | Receive enable |

# Functional Description

This section describes the steps required by the StrongARM core driver to initialize the microengines and to transmit and receive packets.

## Initialization

Before the StrongARM driver begins initializing the interfaces, the microcode must already have been loaded and started. The following steps will then finish initializing the microcode interfaces.

Step 1. Initialize the queue pointers in the IXP1200MM data structure. The 3 push/pull stacks are initialized by pushing a NULL entry, and the 2 linked list queues are initialized by setting the head and tail with each others address.

Step 2. Allocate buffer descriptors. The buffer descriptors should be allocated from SRAM starting at offset 0x80000. Either the transmit or receive descriptors can be allocated first, with the other following immediately in memory.

Step 3. Clear event register. The event register is cleared by writing zeroes to it.

Step 4. Set mask register. Set the appropriate bits in the mask register to enable the interrupts that will be serviced.

Step 5. Write the address of the IXP1200MM data structure to the first 4 bytes of scratchpad memory and signal the microcode to initialize.

Step 6. Signal the microcode interface thread. The interface thread will then start the remaining microcode threads. The interface thread is signaled by writing the thread number (16) to the `INTER_THD_SIG` register at address `0xB00401C0`.

Step 7. Configure the port specific CSRs for the interface being initialized.

Step 8. Configure any required hardware registers for the interface.

Step 9. Enable interrupts. The microcode is again notified of the pending command by signaling the interface thread (thread 16), as done in step six.

Step 10. Send command to start the transmitter and receiver for the port being initialized. This is done by setting the transmit and receive enable bits in the transmit/receive control register (trcr). Next, send the command to initialize the transmit and receive parameters to the microcode. This is done via the microcode command register (mccr) by setting the appropriate port bit and setting the opcode to 0. The microcode is again notified there is a command pending by writing a 16 to address 0xb00401c0.

## Transmit

There are two basic phases required to transmit a packet. The first phase involves passing the data to the microengines. The second phase is required to clean up after the packet has been sent on the wire.

Step 1. Retrieve a buffer descriptor from the `txfree_pull` stack and initialize it. The required fields are `bd_status`, `bd_len`, `bd_data`, and either `bd_port` or `bd_multiport`.

Step 2. Insert the buffer descriptor on the tail of the `txbd` queue. The microcode will automatically start processing buffer descriptors as they are queued. The StrongARM core driver can now return and the final processing will happen in the transmit interrupt routine.

Step 3. Process transmit interrupt. Before generating the interrupt the microcode will have placed the finished buffer descriptor in the `txdone` push/pull stack. The StrongARM driver needs to pull it from this stack, free any associated system memory, and push it on the `txfree` stack.

## Receive

When a packet arrives at an interface, the microengines remove a buffer descriptor from the `rxfree` push/pull stack and begins filling it with data. After the packet reception has completed, the status information is updated and the buffer descriptor is placed in the `rxbd` queue. If requested, an interrupt to the StrongARM core is also generated. If the StrongARM core driver does not process packets fast enough and the microengines run out of available receive descriptors, all incoming packets are simply dropped until descriptors become available.

The steps required to process are receive interrupt are as follows:

Step 1. Remove the receive buffer descriptor from the `rxbd` queue.

Step 2. Extract the data pointer from the receive buffer descriptor and perform whatever processing is required.

Step 3. Allocate a new buffer to receive another packet and update the buffer descriptor fields.

Step 4. Push the buffer descriptor onto the `rxfree` push/pull stack.

# Understanding Queue Functionality

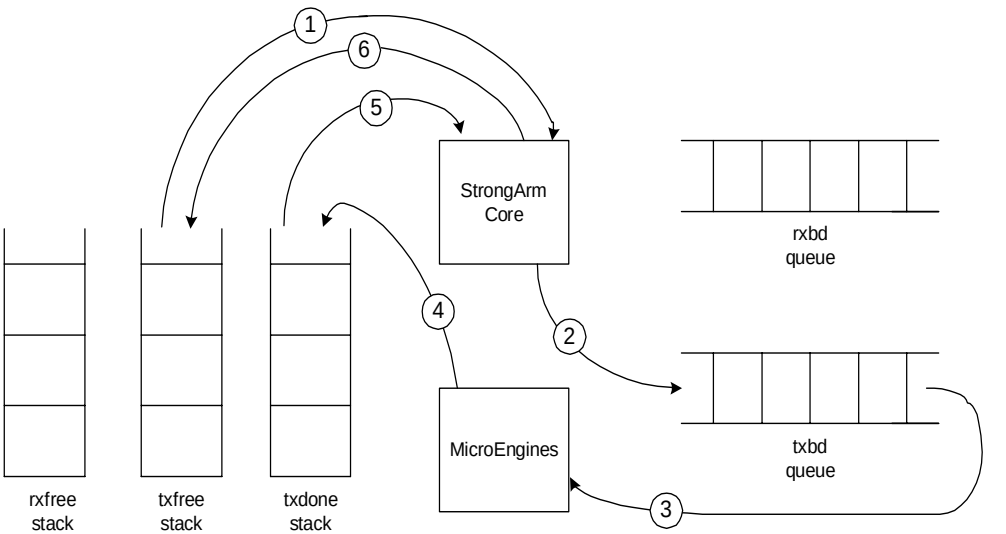
The following sections provide an overview of IXP1200 queue usage and functionality, including the procedures for transmitting, receiving, and forwarding data.

# Transmitting Data

The following steps detail the procedure used to transmit data. This procedure is also illustrated in **Figure 1-11**. (Each step below corresponds with a number in the figure.)

1. The StrongArm core retrieves a buffer descriptor from the `txfree` stack, initializes it, and attaches a buffer to it.
2. The core inserts the buffer descriptor on the tail of the `txbd` queue.
3. The microcode removes a buffer descriptor from the head of the `txbd` queue and processes (transmits) the data.
4. The microcode processes the data, then pushes the buffer descriptor to the `txdone` stack, which interrupts the core.
5. The core takes a buffer descriptor from the `txdone` stack, therby freeing any system memory associated with the buffer descriptor.
6. The core pushes the buffer descriptor on the `txfree` stack.

**Figure 1-11 Transmitting Data**

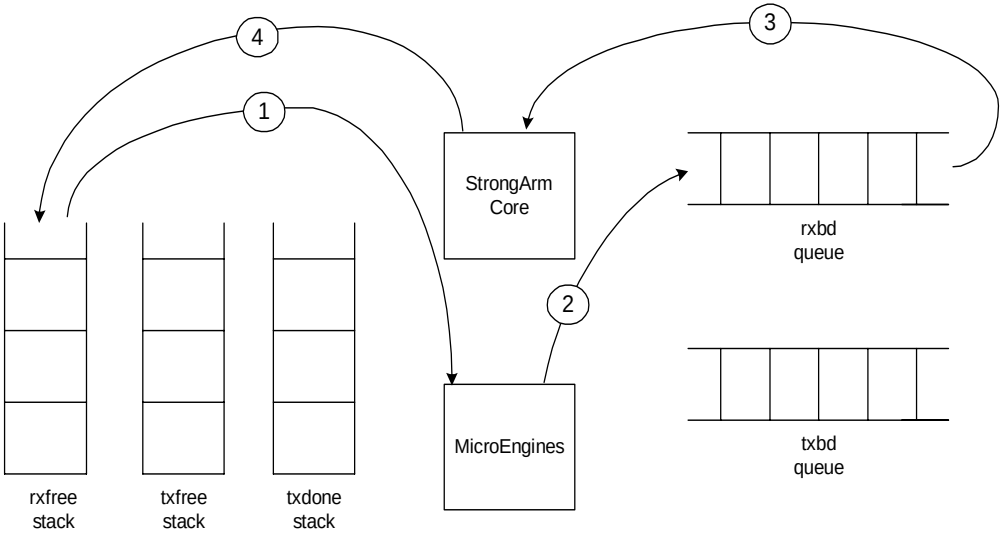


# Receiving Data

The following steps detail the procedure used to receive data. This procedure is also illustrated in **Figure 1-12**. (Each step below corresponds with a number in the figure.)

1. The microcode takes a buffer descriptor from the `rxfree` stack and writes the received data to the data pointer in the buffer descriptor.
2. The microcode inserts the buffer descriptor at the tail of the `rxbd` queue and interrupts the core.
3. The core removes a buffer descriptor from the `rxbd` queue, extracts the memory pointed to by the data pointer, and processes the extracted data (sends the data to the core receiver process).
4. The core allocates a new buffer, attaches the new buffer to the buffer descriptor, and pushes the buffer descriptor to the `rxfree` stack.

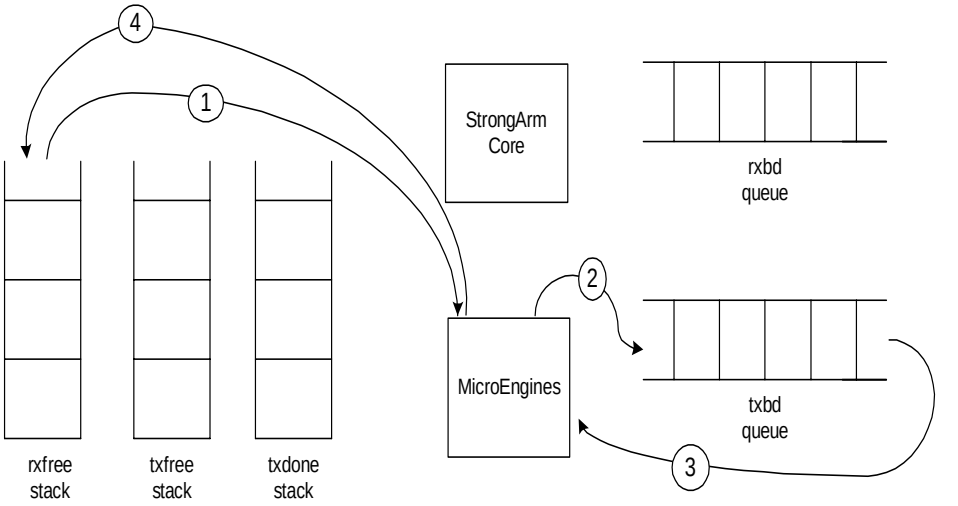
**Figure 1-12 Receiving Data**



# Forwarding Data

The following steps detail the procedure used to transmit data without using the StrongArm core. This procedure is also illustrated in **Figure 1-13**. (Each step below corresponds with a number in the figure.)

**Figure 1-13 Forwarding Data**




1. The microcode takes a buffer descriptor from the `rxfree` stack and writes the received data to the memory pointed to by the data pointer in the buffer descriptor. The TYPE bit in the control/status word of the buffer descriptor is notified that this is a receive buffer.
2. The microcode inserts the buffer descriptor at the tail of the `txbd` queue.
3. The microcode removes the buffer descriptor from the head of the `txbd` queue and processes (re-transmits) the data.
4. The microcode checks the TYPE bit in the buffer descriptor (this bit should be 1) and pushes the buffer descriptor to the `rxfree` stack.

# Appendix A: Data Structures and Layout

This appendix contains the 'C' language definitions for the data structures used by the microcode core drivers. The following sections are included:

- **ICD Data Structures**

# ICD Data Structures

The following code lists the 'C' language definitions for the data structures used by the blocks of memory shown in **Figure A-1**, **Figure A-2**, and **Figure A-3**.

## For More Information

For more information about these data structures, refer to Chapter 1: IXP1200 Microcode/StrongARM Core Interface Control.

```
/* Buffer descriptors */
typedef struct BufferDescriptor {
    struct BufferDescriptor *bd_next;
    u_int16 bd_status;
    u_int16 bd_port;
    u_int16 bd_len;
    u_int16 reserved1;
    u_int16 bd_size;
    u_int16 reserved2;
    u_int8 *bd_data;
    u_int32 bd_usr;
    u_int32 bd_multiport;
    u_int32 reserved3;
} BD;

/* Ethernet interfaces */
typedef struct {
    /* Port Specific Structures */
    u_int8  paddr[6];   /* Unicast address */
    u_int8  res[2];     /* Padding for alignment */
    BD      oobtxbd;    /* Out of band transmit BD */
   u_int32 retry_lmit; /* Max number of collisions allowed 10/100 only
*/
```

```
    /* Port Specific Registers */
    u_int32 trcr;       /* Transmit/Receive Control Register */
} Enet_port;

typedef struct {
    /* Common data structures */
    u_int32 rxfree_push;    /* IXP Push Queue for Empty Receive BD's */
    BD      **rxfree_pull;  /* IXP Pull Queue for Empty Receive BD's */
    u_int32 txfree_push;    /* IXP Push Queue for Empty Transmit BD's
*/
    BD      **txfree_pull;  /* IXP Pull Queue for Empty Transmit BD's */
    u_int32 txdone_push;    /* IXP Push Queue for Transmitted BD's */
    BD      **txdone_pull;  /* IXP Pull Queue for Transmitted BD's */
    BD      *rxbd_tail; /* Tail of receive queue */
    BD      *rxbd_head; /* Head of receive queue */
    BD      *txbd_tail; /* Tail of transmit queue */
    BD      *txbd_head; /* Head of transmit queue */
    u_int32 vector;     /* */

    /* Common Registers */
    u_int32 i_event;    /* Interrupt Event Register */
    u_int32 i_mask;     /* Interrupt Mask Register */
    u_int32 mccr;       /* Microcode Command Register */
} Enet_common;

/* IXP 1200 Shared Memory Map */
typedef struct {
    Enet_common enet_common;
    Enet_port enet_port[MAX_PORTS];
} IXP1200MM;
```

# IXP1200 Memory Usage

The following sections explain how microcode organizes its external memory.

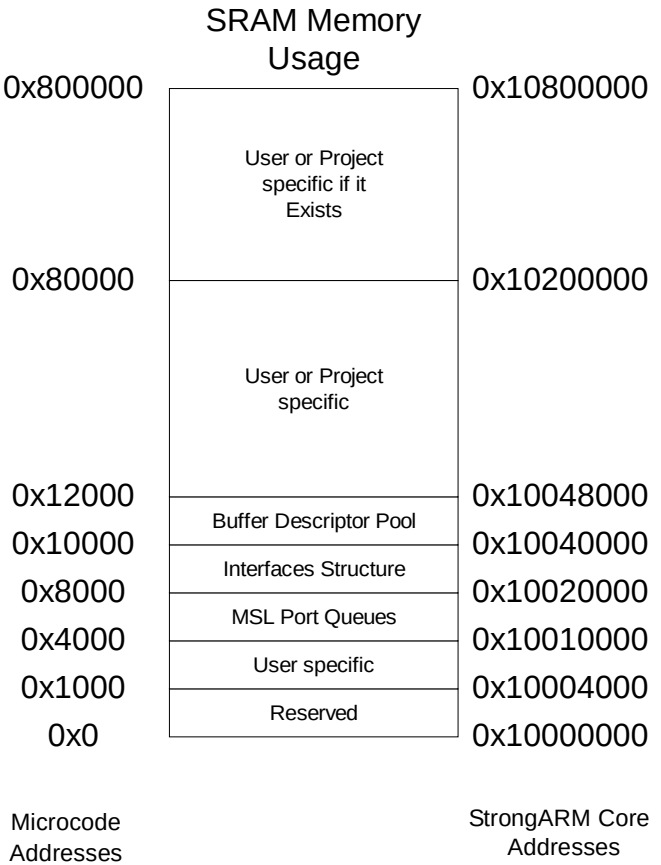
## SRAM

The SRAM is used for operations requiring quick access, including buffer descriptors, lookup tables, data queues, and interface structures. The contents of the SRAM include, but are not limited to, the following elements: (These elements are diagramed in **Figure A-1**.)

| | |
|---|---|
| Buffer Descriptor Pool | Buffer descriptors are allocated from this area. Pointers to this memory are passed on the queues and stacks in order to pass buffers to and from the StrongArm core or to and from the MicroEngines. |
| Interfaces structure | The interfaces structure starts with the IXP1200MM. |
| MSL Port Queues | This memory is used exclusively by the MicroEngines to pass data to a port's transmit queue. |

| | |
|---|---|
| User or Project memory | Memory with this label is used at a project's discretion. For more information regarding the memory used by a particular project, refer to the MSL documentation. |

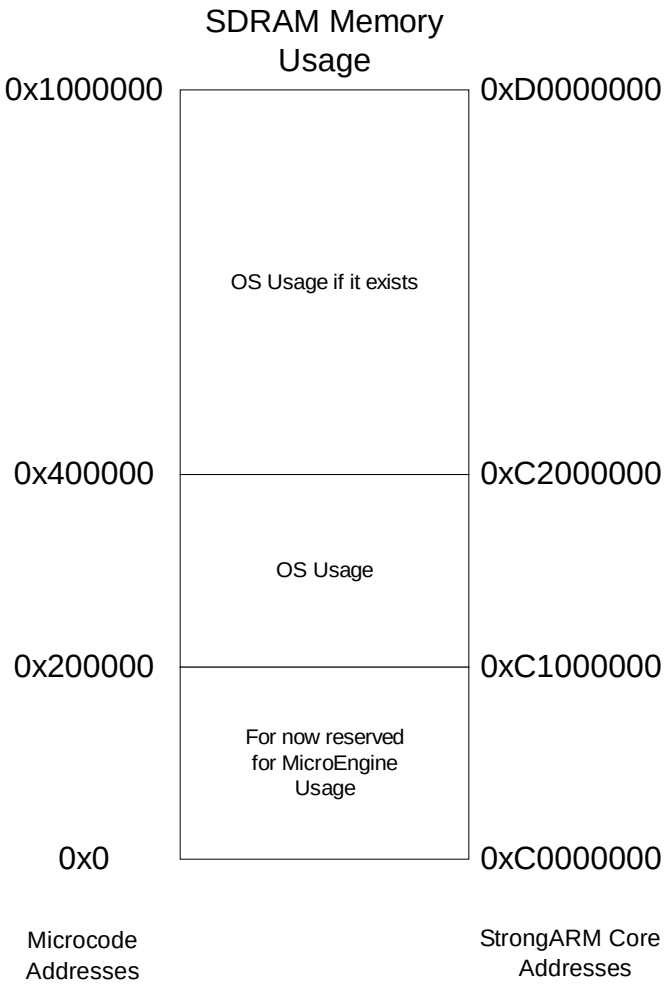
**Figure A-1 SRAM Memory Usage**



## SDRAM

The SDRAM is primarily used for storage not frequently referenced by the microcode. The SDRAM is usually dedicated to the host operating system. Most of the memory required by the microcode is allocated by the host system's memory allocator and handed to the microcode through pointers in the SRAM structures.

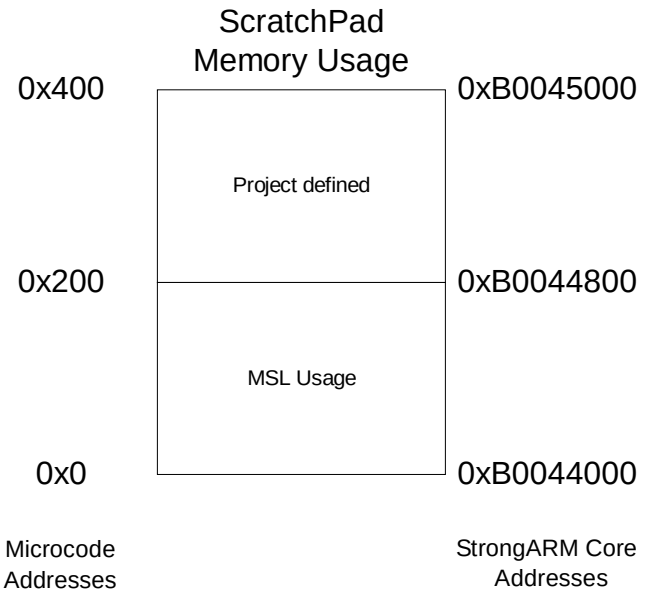
**Figure A-2 SDRAM Memory Usage**



## ScratchPad

The ScratchPad is useful in interthread communication and thus is reserved for the microcode's use. For the MSL, the first two words of the ScratchPad are used as pointers. The first word points to the Interface Structure in SRAM and the second word points to the forwarding table's control block.

**Figure A-3 ScratchPad Memory Usage**



# Product Discrepancy Report

To: Microware Customer Support

FAX: 515-224-1352

From:_____

Company:_____

Phone:_____

Fax:_____Email:_____

Product Name:

Description of Problem:

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Host Platform_____

Target Platform_____

RadiSys.

MICROWARE SOFTWARE