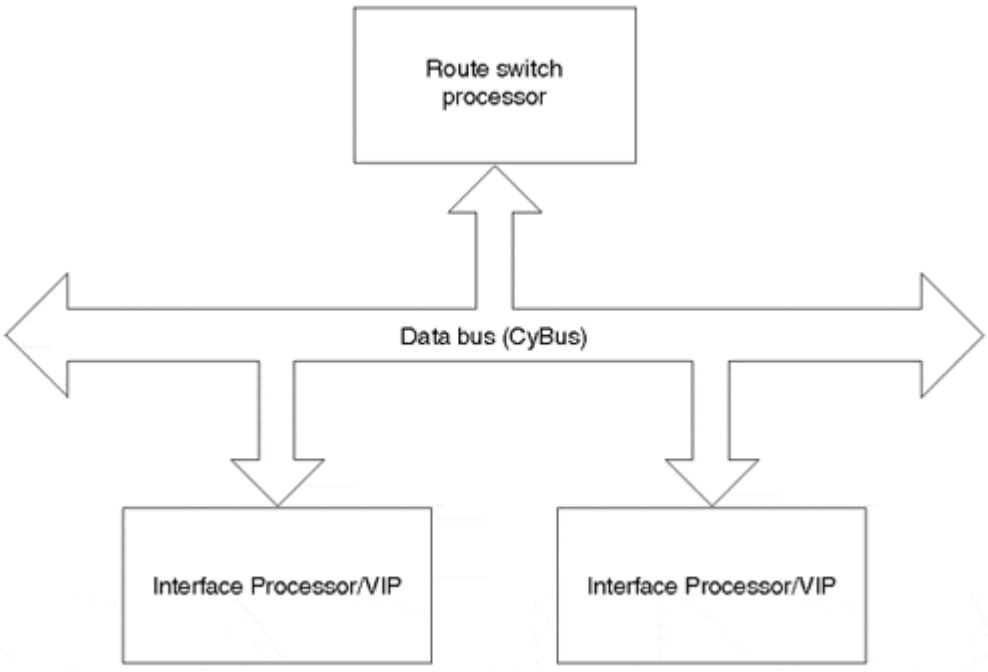
Hardware Architecture of the Cisco 7500 Router

Figure 6-1 provides a high-level diagram of the 7500 router architecture. As you can see from this diagram, there are two components central to understanding the unique features of IOS on the 7500: the data bus and *Route Switch Processor (RSP)*.

Figure 6-1. The 7500 Architecture



The Data Bus

The data bus in the 7500 router, like that of its predecessors (the AGS+ and 7000), is based on the proprietary Cbus architecture.The Cbus used in the 7500 is a faster version of the 7000's Cbus called the *CyBus* (the 7000's Cbus is often called the *CxBus*). Like the original Cbus, the CyBus is a 32-bit bus with 32 data lines, 8 command lines, 24 address lines, and other lines used for control operations such as bus access, acknowledging successful transactions, and error reporting.

Thirty-two lines of data might seem small in comparison to other modern buses, some of which use 128 or more data lines, but this number was chosen so older CxBus interface processors would still work when connected to a CyBus. Newer CyBus-enabled interface processors are capable of performing two transactions (two 4-byte reads) during each bus clock cycle (60 ns/16.67 MHz), while CxBus interface processors can perform only one transaction (one 4-byte read) in each clock cycle. As a result, the CyBus offers 1.066 Gbps (64 bits * 16.67 MHz) of bandwidth, effectively doubling the bandwidth offered by the 533-Mbps CxBus (32 bits * 16.67 MHz).

NOTE

When a combination of CyBus- and CxBus-capable interface processors are present on the CyBus, CyBus-capable interface processors can use the full 1.066-Gbps bandwidth. The CxBus-capable interface processors are still restricted to 533 Mbps.

There are four different router models within the 7500 family, as follows:

- **7505—**

  Single CyBus and five slots (one RSP slot)

- **7507—**

  Dual CyBus and seven slots (two RSP slots)

- **7513—**

  Dual CyBus and 13 slots (two RSP slots)

- **7576—**

  Two independent 7500 routers in the same chassis, each with dual CyBusses

A 7500's model number can be determined by executing **show environment all** on the router's console. Example 6-1 illustrates the output from a 7505 and 7507 router.

Example 6-1. show environment all Command Output on 7505 and 7507 Routers

7505**#show environment all** Arbiter type 1, backplane type 7505 (id 1) …. 7507**#show environment all** Arbiter type 1, backplane type 7507 (id 4) ….

Route Switch Processor

The Route Switch Processor, more commonly called the RSP, performs centralized routing and switching functions (what a surprise!). The RSP is often considered the heart of a 7500 series router because it makes switching decisions, runs routing protocols, and performs maintenance tasks.

The RSP is essentially equivalent to a combined Route Processor (RP) and Switch Processor (SP) from the Cisco 7000 router. Combining the RP and SP function on a single board eliminates the relatively slow Multibus (155 Mbps) needed on the 7000 and AGS+ to connect the RP and SP systems together. This is one of the reasons why an RSP outperforms an RP/SP so dramatically.

Table 6-1 shows the four different RSP types supported by Cisco 7500s.

Table 6-1. Cisco 7500 Router RSP Types

RSP	Characteristics
RSP1	MIPS R4600 or R4700 CPU; single CyBus interface; can be used only on a 7505; Max 128 MB DRAM; 2 MB MEMD
RSP2	MIPS R4600 or R4700 CPU; dual CyBus interface; can be used on a 7505, 7507, 7513, or 7576; Max 128 MB DRAM; 2 MB MEMD
RSP4	MIPS R5000 CPU; dual CyBus interface; can be used on 7507, 7513, or 7576; Max 256 MB DRAM; 2 MB MEMD
RSP8	MIPS R7000 CPU; dual CyBus interface; can be used on 7507, 7513, or 7576; Max 256 MB ECC DRAM; 8 MB MEMD

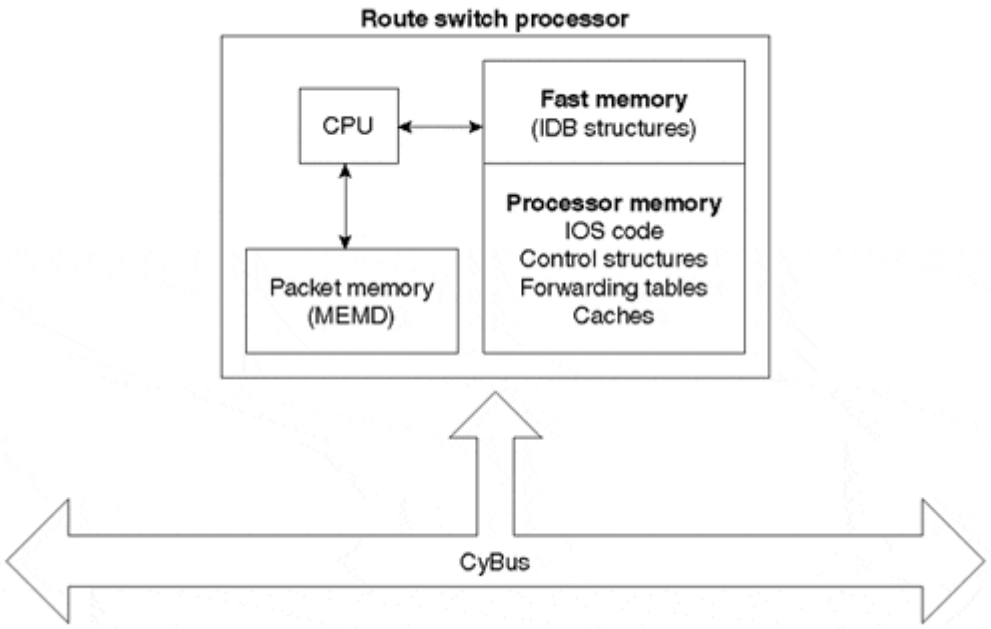
The type of RSP installed in a 7500 can be determined using **show version**, as demonstrated in Example 6-2

Example 6-2. Determining the RSP Type Used on a 7500 Router

router**#show version** cisco RSP2 (R4700) processor with 32768K/2072K bytes of memory. R4700 CPU at 100Mhz, Implementation 33, Rev 1.0 ….

Figure 6-2 illustrates a high level overview of the RSP.

Figure 6-2. A Route Switch Processor



Three of these components play a major role in how IOS—in particular, how IOS switching—operates on the 7500 router.

- CPU
- Fast packet memory (designated by MEMD)
- Main memory (designated by DRAM)

The following sections describe these components in more detail.

CPU

The RSP's CPU is the main processor on the 7500 router; it's responsible for running IOS and switching packets. Unlike the 7000 and AGS+, there's no separate packet switching microcode (SP microcode or Cbus controller microcode); with the exception of distributed switching, all switching decisions on the 7500 are performed in IOS running on the RSP CPU. RSPs use MIPS Reduced Instruction Set Computer (RISC) processors; the clock rate, the cache size, and the type of cache depend on the model.

Fast Packet Memory

RSP1s, RSP2s, and RSP4s have a fixed 2 MB bank of SRAM reserved for packet buffers; the RSP8 has 8MB of SDRAM reserved for packet buffers. This fast packet memory region is named *MEMD* just like the similar region in the Cisco 7000 and AGS+. The 7500's MEMD is connected to both the RSP CPU and the interface processors (via the CyBus); both have direct access to it as a shared resource.

Although MEMD is carved into pools of buffers using an algorithm similar to the one used on the Cisco 7000, some changes in the hardware allowed improvements to be made. For example, the 7500 can carve up to 3520 buffers, a vast improvement over the 7000's limit of 470. A complete description of the algorithm is beyond the scope of this book, but a simplified overview of the process follows:

Step 1. Interfaces on the router are classified into various groups based on MTU size. The range of MTU sizes accepted in each group is initially set to a variance of 256 bytes. If there are more MTU sizes than can be accommodated by four groups, the variance is doubled to 512 bytes. If there are still more than four

groups, the variance is doubled again to 1024, and so on until at most four groups are defined.

Step 2. Each of the buffer pools from Step 1 gets 20 percent (360 KB) of MEMD, the remaining MEMD is then divided among the pools based on the aggregate bandwidth of all interfaces within a pool.

To illustrate the process, assume there are five interfaces on a router with MTUs of:

- 512 bytes
- 1024 bytes
- 1500 bytes
- 4096 bytes
- 16,000 bytes

IOS begins by defining five packet buffer pools:

- **poolA—**

  512 byte buffers, serves interfaces with MTU in the range 256 < MTU = 512 bytes

- **poolB—**

  1024 byte buffers, serves interfaces with MTU in the range 768 < MTU = 1024 bytes

- **poolC—**

  1500 byte buffers, serves interfaces with MTU in the range 1244 < MTU = 1500 bytes

- **poolD—**

  4096 byte buffers, serves interfaces with MTU in the range 3840 < MTU = 4096 bytes

- **poolE—**

  16,000 byte buffers, serves interfaces with MTU in the range 15,744 < MTU = 16,000 bytes

Because there are more than four pools, the algorithm tries again using the 512 byte variance and comes up with new pool definitions:

- **poolA—**

  512 byte buffers, serves interfaces with MTU in the range 0 < MTU = 512 bytes

- **poolB—**

  1500 byte buffers, serves interfaces with MTU in the range 988 < MTU = 1500 bytes

- **poolC—**

  4096 byte buffers, serves interfaces with MTU in the range 3584 < MTU = 4096 bytes

- **poolD—**

16,000 byte buffers, serves interfaces with MTU in the range 15,488 < MTU = 16,000 bytes

This time, all five interfaces can fit into four pools. Notice poolB now accommodates interfaces with the 1024- and 1500-byte MTUs.

It's rare there would be more than four MTU sizes on a given router, so this process generally doesn't iterate further than the first step. If there are less than four MTU sizes configured across the interfaces, IOS creates only the number of pools needed.

After the pool sizes have been decided, IOS divides MEMD into five parts, each representing 20 percent of the total memory available. One part is placed in each of the four pools, and the remaining part (20 percent) is divided among the pools based on the types and speeds of interfaces associated with each pool. There is some overhead associated with MEMD control structures—for example, some MEMD is reserved for communication between the CPU and the interface processors—so the entire 2 MB of MEMD is never available solely for packet buffers.

In the preceding example, we assumed the MEMD buffer sizes were equal to the MTU in each buffer pool for simplicity, but in practice, MEMD buffers are created slightly larger than the largest MTU accommodated by a pool. MEMD buffers must be large enough to hold the largest MTU and still have room for an encapsulation header to be added on. Beyond this, MEMD buffers must be an even multiple of 32 bytes so they align properly in the hardware memory cache for optimum performance. As a result, MEMD buffers receive a memory allocation based on the formula *size* = *mtu* + *e* + *n*, where *mtu* equals the largest MTU, *e* equals the largest encapsulation that can be added for the associated interface, and *n* is the number of filler bytes (0–31) required to make the result an even multiple of 32.

NOTE

The MEMD buffer carving algorithm does not take into account the state of an interface (that is, whether it is up or down) when deciding what size pools to create. Buffers are allocated to unused and administratively down interfaces just as if they were up and running full traffic. The rationale is those administratively down interfaces could still be turned on at a later time and require the use of MEMD resources, so they have to be taken into account. Because the presence of unused interfaces can result in thinner distribution of MEMD buffers overall, it's best to remove any unused interface processors, if possible, to make the maximum amount of MEMD available to the active interfaces.

MEMD resources are used most efficiently if all interfaces are configured for a common MTU size (typically 1500 bytes). That way, the MEMD carving algorithm creates only one buffer pool shared among all the interfaces.

The **show controller cbus** command output provides a snapshot of MEMD control structures and shows how it has been carved—invaluable information for troubleshooting. Example 6-3 shows a partial output from the command followed by descriptions of some interesting fields.

Example 6-3. show controller cbus Command Output

router#**show controller cbus** MEMD at 40000000, 2097152 bytes (unused 2976, recarves 4, lost 0) RawQ 48000100, ReturnQ 48000108, EventQ 48000110 BufhdrQ 48000128 (2939 items), LovltrQ 48000140 (11 items, 2016 bytes) IpcbufQ 48000150 (16 items, 4096 bytes) IpcbufQ_classic 48000148 (8 items, 4096 bytes) 3570 buffer headers (48002000 - 4800FF10) pool0: 9 buffers, 256 bytes, queue 48000130 pool1: 278 buffers, 1536 bytes, queue 48000138 pool2: 305 buffers, 4544 bytes, queue 48000158 pool3: 4 buffers, 4576 bytes, queue 48000160 slot1: EIP, hw 1.5, sw 20.06, ccb 5800FF30, cmdq 48000088, vps 4096 software loaded from system Ethernet1/0, addr 00e0.8f6d.a820 (bia 00e0.8f6d.a820) gfreeq 48000138, lfreeq 48000168 (1536 bytes), throttled 015 rxlo 4, rxhi 101, rxcurr 0, maxrxcurr 1 txq 48000170, txacc 48000082 (value 49), txlimit 50

- **MEMD at 40000000, 2097152 bytes (unused 2976, recarves 4, lost 0)—**

  This line shows MEMD begins at address 0x40000000 in the CPU's memory and is 2 MB. The

unused field reports the number of unused bytes—this is usually small (less than the largest MTU) and is nothing to worry about. The **recarves** field indicates the number of times the IOS MEMD carving algorithm has run. Several situations can cause a MEMD recarve, including the insertion or removal of a line card, changing the MTU on an interface, or a microcode reload. This field is set to 1 when the initial MEMD carve is run during IOS initialization.

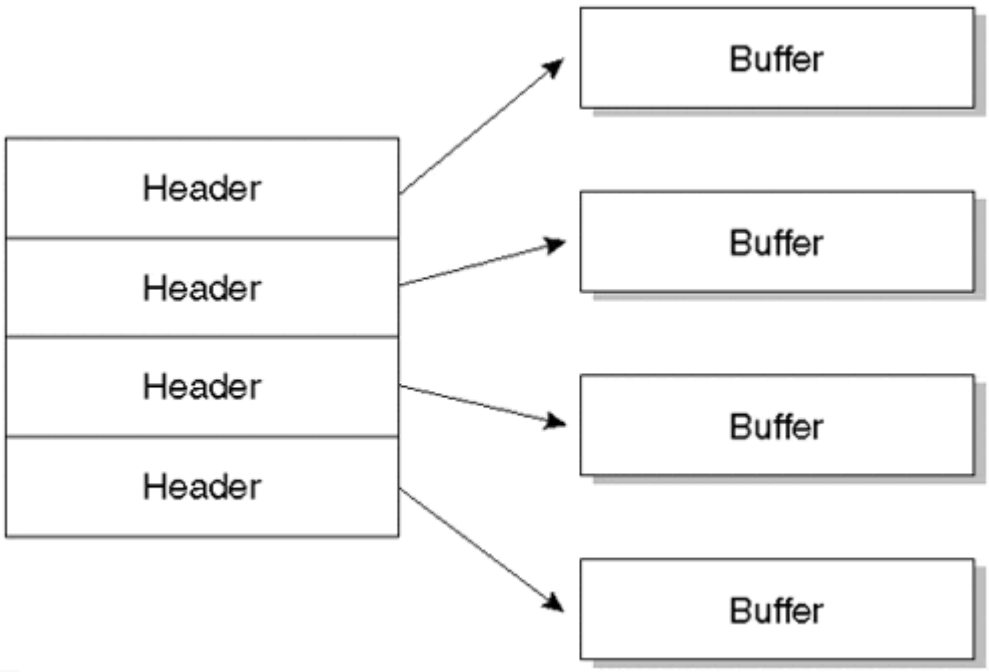
- **RawQ 48000100, ReturnQ 48000108, EventQ 48000110BufhdrQ 48000128 (2939 items), LovltrQ 48000140 (11 items, 2016 bytes) lpcbufQ 8000150 (16 items, 4096 bytes)lpcbufQ_classic 48000148 (8 items, 4096 bytes)—**

  These lines show the memory addresses of various internal MEMD queue data structures. These data structures represent lists of MEMD buffers (actually, buffer headers, which are described in the following bullet), some of which are used in processing packets. Of particular interest is the one called *RawQ*. The RawQ, as described later in the section on packet switching, is used to queue all packets received by interface processors to the CPU on the RSP.

- **3570 buffer headers (48002000 - 4800FF10)—**

  This line shows the total number of addressable MEMD *buffer headers*. A buffer header is a control structure containing data about a MEMD buffer and a pointer to the actual location of the buffer (see Figure 6-3). Every MEMD buffer has its own unique buffer header.

Figure 6-3. Buffer Headers



These buffer headers are a lot like labels on a soup can; they provide a quick snapshot of what's inside without having to open the can. The headers provide a convenient way for IOS to pass around descriptive information about MEMD packet buffers without having to send the buffers themselves. On the RSP there can be a total of 3570 buffer headers.

- **pool0: 9 buffers, 256 bytes, queue 48000130 pool1: 278 buffers, 1536 bytes, queue 48000138 pool2: 305 buffers, 4544 bytes, queue 48000158 pool3: 4 buffers, 4576 bytes, queue 48000160—**

  These lines show how the available 2 MB of MEMD has been carved. The first buffer pool (**pool0**) and the last pool (**pool3** in this example) are reserved for interprocess communication between the interface processors and the RSP CPU, while the other two pools, **pool1** and **pool2**, are carved for packet buffers.

Packets are stored in MEMD buffers based on the interface they are received on, not the size of the packet. For example, if a 64-byte packet is received on an interface with a 1500-byte MTU (1536-byte MEMD buffer size), it still is stored in a 1536-byte buffer—even if another pool exists with smaller buffers.

If a packet arrives on an interface and there no available MEMD buffers in the interface's pool, the packet is dropped and the interface's **ignored** counter is incremented. This situation occurs even if there are buffers available in other pools. Setting all the interface MTUs to the same size results in all MEMD buffers being placed in one pool, which reduces the likelihood of a packet being dropped because the interface pool is empty.

- **slot1: EIP, hw 1.5, sw 20.06, ccb 5800FF30, cmdq 48000088, vps 4096software loaded from system—**

  This shows data about the interface processor in slot 1; each slot populated with an interface processor will have similar information displayed. An Ethernet interface processor (**EIP**) is in slot 1, its hardware version (**hw**) is 1.5, and the software version (**sw**) is 20.06. The **ccb** and **cmdq** are data structures used for communicating with the RSP.

- **Ethernet1/0, addr 00e0.8f6d.a820 (bia 00e0.8f6d.a820gfreeq 48000138, lfreeq 48000168 (1536 bytes), throttled 015rxlo 4, rxhi 101, rxcurr 0, maxrxcurr 1txq 48000170, txacc 48000082 (value 49), txlimit 50—**

  These lines provide further information about each interface, including pointers to the locations of their various MEMD data structures:

  - **gfreeq** is a pointer to the *global free queue*. The global free queue is a list of all the free MEMD buffers in a particular pool. All interfaces sharing the same MEMD buffer pool point to the same global free queue.

  - **lfreeq** is a pointer to the interface's private *local free queue*. In addition to the global list of free MEMD buffers in each pool, individual interfaces can keep a local "stash" of free buffers obtained from the global pool. While all free buffers initially come from the global free queue, once an interface has used a buffer, it can keep the buffer for a short period of time. This helps prevent ignored packets when there are other busy interfaces competing for buffers in the global free pool. Unused free buffers on the local free queue are periodically returned to the global free queue for use by other interfaces.

  - **rxhi** shows the maximum number of MEMD buffers a particular interface can hold at any time. This prevents one interface from holding all the free buffers in the global pool and starving other interfaces.

  - **rxcurr** represents the number of buffers currently held by this interface (this counter may constantly be non-zero for a very busy interface).

  - **maxrxcurr** represents the maximum number of buffers this interface has ever held since the last time MEMD was recarved—that is, it's a "high water" mark. This number should never be higher than the *rxhi* value.

  - **rxlo** is always 4. This field indicates the smallest number of buffers this interface will hold in its local free queue when idle (assuming there was ever traffic on this interface).

  - **txq** is a pointer to a list of buffers containing packets waiting to be transmitted out this interface. The list is called the *transmit queue* and this field is known as the *transmit queue pointer*.

  - **txlimit** is the maximum number of buffers allowed on the interface's transmit queue at any time. The value in parentheses preceding this field shows the remaining number of buffers allowed on the transmit queue. In Example 6-3, the value in the parentheses is 49 and the

txlimit is 50. This means that one packet is currently enqueued on the interface's transmit queue. A value of zero within the parentheses indicates no more packets can be placed on the transmit queue until some are removed.

Main Memory

Main memory on an RSP consists of up to 128 MB (256 MB on RSP4s and RSP8s) of upgradable DRAM. In general, RSP main memory contains the IOS code plus all the data structures IOS uses at runtime. Example 6-4 displays the output from **show memory summary** on a Cisco 7500 router.

Example 6-4. show memory summary Command Output for 7500 Router

router#**show memory summary** Head Total(b) Used(b) Free(b) Lowest(b) Largest(b) Processor 60AD9EE0 55730464 9405352 46325112 45891060 46251636 Fast 60AB9EE0 131072 82168 48904 48904 48860 ….

From this example, you can see IOS divides RSP main memory into two memory pools: *processor* and *fast*. The fast memory pool is used to store interface data structures called *interface descriptor blocks,*while the processor pool is used to store instructions, data, control structures, packet forwarding tables, system packet buffers, and the heap. Although it might appear limited, the small amount of memory in the fast memory pool is not a constraining factor; if the number of interface descriptor blocks exceeds the available fast pool, IOS simply begins using memory from the processor memory pool.

Processor memory on the 7500 platform also contains the system buffer pools, which are described in Chapter 1, "Fundamental IOS Software Architecture."

< BACK Make Note | Bookmark CONTINUE >

Index terms contained in this section