# Port Card Controller Specification

Version 2-0

Cliff Young
Harvard University

(617) 496-4513
cyoung@das.harvard.edu
Pierce Hall 110C, 29 Oxford Street, Cambridge, MA 02138

This document specifies the Port Card Controller microprocessor on the Port Card.

## 1.0 Introduction

### 1.1 Purpose/Description

The PCC is the catch-all management microprocessor on the Port Card. It initializes and tests all the other port card components; it establishes, modifies, and removes VCs, it gathers some statistics, it performs coherence checking on port card data structures, and it assists the Scheduler with slow-response functions.

### 1.2 Requirements

1. The PCC must interface to all the necessary blocks, registers, memories, and processors on the Port Card to ensure proper initialization, operation, and debugging.
2. The PCC must bootstrap its initial code from the MCP.
3. The PCC must test all memories and blocks on the port card before operations begin.
4. The PCC must initialize all memories and configuration registers before operations begin.
5. In all operating modes, the PCC must be able to sample the status and error counter (readable) registers of all the hardware blocks on the port card.
6. In normal operations (henceforth called "Run Mode"), the PCC must be able to set up, tear down, and reprioritize connections.
7. The PCC must be able to force the port card into "Single-Step Mode", where the port card will only send or receive a cell under PCC/MCP direction.

8. If all port cards are in Single-Step Mode, and no ports are sending or receiving cells under PCC/MCP control, then the state of all internal hardware blocks must remain unchanged.
9. The PCC must be able to engage and disengage all implemented loopback paths on the port card.
10. The PCC must be able to initiate a "Recovery Mode", similar to Single-Step Mode, where no cell input or output takes place and all data structures are available and consistent for examination.

## 1.3 Functions Supported

The port card probably runs in a number of interesting operating modes:

1. Initialization

   This mode occurs only at switch start-up. The PCC should test all memories, download code, set initial configurations, and enter one of Run, Debug, or Single-Step Modes.
2. Debug Monitor

   This piece of software can be invoked by the MCP and possibly by a physical switch on the port card (to allow invocation on a stand-alone port card). This mode provides an assembly-level monitor for low-level debugging of PCC code.
3. Run

   This is the normal operating mode for the switch. In this mode, the PCC must set up VCs, tear them down, modify existing VCs, gather some statistics, and poll hardware status registers.
4. Single Step Mode

   This mode allows the switch to send and receive one cell per port per "single-step" signal. Between single cells, the entire state of the switch is visible.
5. Loopback Mode

   This mode allows the port card to activate any of its hardware loopback paths.
6. Recovery Mode

   This mode allows the switch to turn off its inputs, examine and reinitialize its data structures (possibly preserving VCs), and rebegin normal operations. It is not clear at this point whether we need this mode.

## 1.4 Functional Block Diagram

## 1.5 Pseudocode

Please see section 6, Software, for detailed pseudocode.

## 1.6 Timing Diagrams

For the time being, see the i960 data book. Jim McLaughlin's forthcoming design document will include detailed timing diagrams.

### 1.7 VHDL

## 2.0 Interfaces to Other Blocks

### 2.1 MCP and Scheduler

Communication with other switch microprocessors will use shared memory and interrupts. Each shared memory region will support atomic read-modify-write instructions (atadd and atmod in the i960 instruction set). As of this writing, Jim also planned to support address collision resolution, so that if two processors access the same memory address, no garbled data will be read.

The MCP will be able to individually reset each PCC; each PCC will be able to reset its associated Scheduler. There are not currently any plans to allow the MCP to directly reset the Schedulers.

The Harvard team believes that a single interrupt level is necessary from the MCP to the PCC, from the PCC to the MCP, from the PCC to the Scheduler, and from the Scheduler to the PCC. As of this writing, the BNR software team felt more comfortable supporting two levels of interrupts from PCC to MCP and back; this will be resolved between the BNR software and hardware teams.

### 2.2 Hardware Blocks: Registers and Interlocked Memories

All i960 I/O is memory-mapped. When interfacing to special-purpose hardware, there are basically three types of interfaces.

**1.** Registers.
This interface occurs in any computer system.

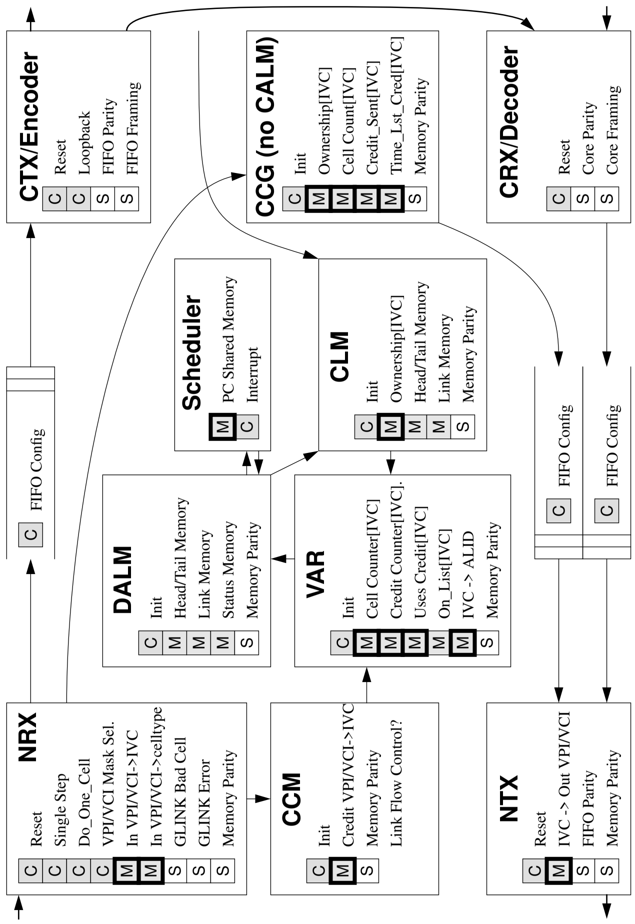
**2.** Interlocked Single-Port SRAMs.
Few of the blocks use all of their memory bandwidth. By adding interlock circuitry, the PCC can access these kinds of memories during idle cycles.

**3.** High-Contention Memories
For high-contention memories (e.g. cell counters in the VAR), the hardware will probably provide only one or two slots in the fixed bus cycle schedule for PCC access. Additional load and store buffer hardware will be necessary, since the PCC and port card pipeline run at different clock rates.

### 2.3 Diagram of Specific Block Connections

This diagram is out of date. It was last updated for the previous version of this document.

Block Connections




## 2.4 List of Specific Block Connections

This list is out of date. It will be superseded by Jim's final hardware design, which will most likely include the port card memory map assembled by the BNR hardware and software teams. This section will be removed when the memory map is complete.

### 2.4.1 Scheduler

1. Dual-Port Shared Memory Area. All data sent from PCC to Scheduler or vice versa goes through this shared memory.
2. PCC->Scheduler Interrupt Signal. This signal allows the PCC to interrupt the Scheduler (e.g. to enter Single-Step Mode).

3. Scheduler->PCC Interrupt Signal. Conversely, this signal allows the Scheduler to request something of the PCC. We don't have any current examples of why this is needed.

### 2.4.2 Master Control Processor (MCP)

1. Dual-Port Shared Memory Area. Similar to shared space between PCC and Scheduler.
2. MCP->PCC Interrupt Signal.
3. PCC->MCP Interrupt Signal.

### 2.4.3 Network Receiver (NRX)

1. MODE (configuration register). This register sets the mode of the NRX. It can currently be one of ON, OFF,SINGLE_STEP, and [optionally] LOAD_GEN. If the NRX is ON, then it processes incoming cells normally. If it is OFF, then it ignores the incoming line and floats its memory buses. If it is in SINGLE_STEP mode, then on DO_ONE_CELL from the PCC, the NRX waits for the beginning of the next cell, sends exactly one cell, then waits for the next signal. In LOAD_GEN mode, the NRX pads cells beginnings written by the PCC to the Load Generator Stuff Register, below.
   This mode will also include LOOPBACK options once we resolve which kinds of electrical loopback will be supported in the port card.
2. CLOCK_SEL (configuration register). This register tells the NRX whether it should send the recovered clock from the input link to the core for reconstruction into the system clock. This is used for synchronous networks.
3. DO_ONE_CELL (configuration register). This register tells the NRX to accept the cell that begins at the next synchronization point on the input link.
4. VPI/VCI Mask Configuration (3 bits configuration register). This register tells the NRX which subfields of the 28-bit VPI/VCI combination to consider when mapping incoming VPI/VCI fields into IVCs.
5. VPI/VCI -> IVC (64K x 16 interlocked SRAM). This memory maps from incoming VPI/VCI to IVC. It is indexed by the significant parts of the VPI/VCI (as determined by the VPI/VCI Mask Configuration register).
6. Incoming VPI/VCI to cell type map (64Kx4 Interlocked SP SRAM). This map tells the NRX what to do with a cell. Current acceptable values are CREDIT, NOT_CREDIT, NOT_VALID, and RESERVED. Cells with CREDIT VPI/VCI values are sent to the CCM. NOT_CREDIT cells are sent to the core by way of the CTX and Encoder. NOT_VALID cells are sent to the core and are counted as errors. RESERVED is currently unused.
7. NRX to CTX FIFO Configuration (x bit write register). This register allows the PCC to configure the almost-empty and almost-full flags on the programmable FIFO between NRX and CTX.
8. Bad HEC Counter (8 bit status register). Count of how many bad HECs the NRX has encountered since the last sample by the PCC. Maxes at 255; does not wrap.
9. Bad CRC32 Counter (8 bit status register). Count of bad credit cell CRC32s.
10. Short Cell Counter (8 bit status register). Count of short cells padded by the NRX.
11. Long Cell Counter (8 bit status register). Count of long cells truncated by the NRX.

12. Glink Error Counter (8 bit status register). Count of detected GLINK errors.
13. Memory Parity Error Counter (8 bit status register). Count of parity errors on the NRX client memories.
14. [Optional] Load Generator Stuff Register (32 bit configuration register). The PCC writes this register to create the first 8 bytes (5 header + 3 payload) of load-generated cells. The NRX will pad the rest of the cell and send it to the CRX.

### 2.4.4 Core Transmitter (CTX)

1. MODE (Configuration Register). ON or OFF. Will include LOOPBACK controls once we decide which loopback paths will be supported.
2. Framing Error (8 bit read register). Count of incorrectly framed cells coming out of the rate-matching FIFO.
3. FIFO Parity Error (8 bit read register). Count of parity errors on the data from the rate-matching FIFO.

### 2.4.5 Core Receiver (CRX)

1. MODE (Configuration Register). ON or OFF, later including LOOPBACK.
2. IVC to Outgoing VPI/VCI map (64Kx28 Interlocked SP SRAM). This maps IVCs to outgoing VPI/VCI values. It also supports VP working, where only the VPI field is remapped and the VCI field is preserved from the input.
3. CRX to NTX FIFO Configuration (x bit configuration register). Similar to the NRX-CTX rate-matching FIFO configuration.
4. Parity Error (8 bit read register). Count of parity errors on the data from the core.
5. Memory Parity Error (8 bit read register). Count of parity errors on the IVC to outgoing VPI/VCI map.

### 2.4.6 Network Transmitter (NTX)

1. MODE (Configuration Register). ON or OFF, later to include LOOPBACK modes.
2. Credit FIFO Parity Error (8 bit status register). Count of parity errors on the FIFO from the CCG.
3. Data FIFO Parity Error (8 bit status register). Similar to previous.

### 2.4.7 Cell List Manipulator (CLM)

1. MODE (Configuration Register). ON or OFF. This may be the same register for all of the CLM, VAR, DALM, CCG, and CALM, because all of these components operate in synchrony with each other wand with the core.
2. Ownership[IVC] (8Kx1 interlocked SRAM). If the IVCth bit of this table is set, then this CLM will Enqueue the corresponding W_IVC, W_CADR pairs when it sees IVC on the W_IVC bus.
3. Head/Tail Memory (8Kx32 Interlocked SRAM). Head and Tail CADRs, indexed by owning IVC.
4. Link Memory (8Kx16 Interlocked SRAM). Successor of CADR. This means that one cell can belong to no more than one IVC at a time.
5. Memory Parity (Status Register). Count of memory parity errors for this block.

### 2.4.8 VC Activation Requester (VAR)

1. MODE. See CLM.
2. Cell Counter Memory (8Kx8 Interlocked SRAM). Count of how many cells the VC has stored in the CLM.May actually be one less than the number of cells in the CLM, because of synchronization mechanism between VAR, CLM, and Scheduler.
3. Credit Counter Memory (8Kx8 Interlocked SRAM). Count of unused credit for the VC. Similarly to previous, may be one less than actual credit used.
4. Uses Credit Memory (8Kx1 Interlocked SRAM). If this flag is set, the VC is flow-controlled and uses credit. If not, then the VAR always activates the VC on seeing its cells.
5. On_List Memory (8Kx1 Dual-Port SRAM). This flag tells whether the VC is on a list in the DALM.
6. IVC -> ALID (8K x 12 Interlocked SRAM). The VAR looks up the group of an IVC using this memory. The PCC can change a VC's group by writing to this memory.
7. Memory Parity (Status Register). Count of memory parity errors for this block.
8. Chalk mark memory. +++
9. Cell Sent memory. +++

### 2.4.9 Data/Management Active List Manipulator (DALM)

1. MODE. See CLM.
2. Head/Tail Memory (4Kx24 Interlocked SRAM). See CLM, except that Heads and Tails are IVCs in active lists (groups).
3. Link Memory (4Kx12 Interlocked SRAM). See CLM, except that elements are IVCs, and point to the next IVC.
4. Status Memory (128x32 Dual-Port Interlocked SRAM). Indicates whether a given active list is empty or non-empty. Updated by DALM, read by Scheduler.
5. Memory Parity (Status Register). Count of memory parity errors for this block.

### 2.4.10 Credit Active List Manipulator (CALM)

All registers are similar to the DALM.

### 2.4.11 Credit Cell Manipulator (CCM)

1. MODE. Similar to CLM, but separate, since the CCM is not synchronized to the core.
2. Credit Mask Configuration Register. Similar to NRX, but for credit VPI/VCI pairs.
3. Credit VPI/VCI -> IVC (64K x 16 Interlocked SRAM). Maps credit VPI/VCI values to IVCs relevant to this switch.
4. Memory Parity (8 bit status register). Count of memory parity errors for this block.

### 2.4.12 Credit Cell Generator (CCG)

1. MODE. See CLM.
2. Ownership[IVC] (8Kx1 Interlocked SRAM). Similar to CLM ownership map, but for the input side CCG instead of the output side CLM.

3. Cell_Count[IVC] (8Kx8 Interlocked SRAM). Count of cells stored in core memory for this IVC.
4. Credit_Sent[IVC]. I don't know.
5. Time_Last_Credit[IVC] (8Kx32 Interlocked SRAM). Timestamp of the last time credit was sent for the indexed IVC.
6. Memory Parity (8 bit status register). Count of memory parity errors for this block.

### 2.5 Pin Definitions

See the i960CA/CF data book for specific information about i960 pin definitions. Jim's bus design will show detailed pin descriptions.

### 2.6 Interface Timing Definitions and Timing Diagrams

Similar to previous.

## 3.0 Hardware Implementation

### 3.1 Detailed Block Diagram

See Jim's forthcoming work on this topic.

### 3.2 Detailed Timing Diagrams

See Jim's forthcoming work on this topic.

### 3.3 Internal Data Structures (Memory Requirements)

Our original expectation was not to need more than 64K of program store for the PCC software. We now plan to run the VxWorks kernel on the PCC, which by itself occupies 64K. Software filtering data structures may require as much as 256KB of memory, for certain algorithms. See the software filtering document for more details. Because of these anticipated requirements, the software team has requested 512KB of PCC memory.

Note that for software filtering purposes, it is desirable that some portion of the PCC memory be fast, 32-bit, burst- and pipeline-access memory.

### 3.4 Pipeline Description

The i960 is a pipelined microprocessor, but its pipeline design is not our responsibility.

### 3.5 Estimates of chips needed, board space, power

1. Intel i960CF Microprocessor
2. Address Decoder Logic. Standard decode chip set.
3. Interrupt and reset logic.
4. Shared Memories with MCP and Scheduler.

5. Interlock Logic for shared memories.

# 4.0 Block Initialization

## 4.1 Self Initialization

The i960 hardware performs self-testing in ONCE™ mode as power comes on.

See the software description and pseudocode for the software startup sequence.

## 4.2 PCC- or Scheduler-Assisted Initialization

See specifications of other blocks and Gary's System Initialization document for block initialization requirements. Most of the hardware blocks will require the PCC to clear their memories and set certain configuration registers.

# 5.0 Block Diagnostics and Self Test

## 5.1 Self Test Routines

PCC self-check (during bootstrap).

## 5.2 Support for PCC Access

Not applicable.

## 5.3 Single-Step Mode Operation

See below for more details.

# 6.0 Software Structure

This section documents the design of the PCC software. We initially plan to run the VxWorks kernel, a single "primary" thread, and interrupt support. Initially, requests from MCP to PCC will work like Remote Procedure Calls (RPCs), where the PCC can perform only one thing at a time on behalf of the MCP. As we refine the software, we may move to a more complex interaction, allowing multiple requests in either direction between the MCP and PCC.

## 6.1 Runtime System: VxWorks

To simplify switch software development, we believe running the VxWorks kernel on the PCC makes the most sense. The VxWorks kernel supports preemptive multithreading and interrupt handling. With additional portions of the VxWorks operating system, we can also add support for synchronization, inter-process communication, and I/O routines.

Ideally, the PCC software will require only one thread plus interrupt support under the VxWorks kernel. If it turns out that we need more than one thread, VxWorks will allow us to add more threads easily.

### 6.2 Primary Thread

It seems simplest to initially write one thread to perform “normal” PCC work. This thread will have to perform a number of tasks:

1. Polling for MCP requests
2. VC setup and teardown
3. VC maintenance
4. Coherence Checking
5. Software Filtering

   See the Software Filtering document for more detail about interesting software filtering schemes. Software Filtering can provide a number of useful services, including bandwidth monitoring and enforcement, Scheduler support, fast credit allocation, and gathering statistics.

Note that initialization may also be performed by the same thread.

### 6.3 Interrupt Hierarchy

i960s may handle interrupts in one of two ways: traditionally, or through interrupt posting. I think that there are four sources of interrupts to the PCC. By priority (highest priority has lowest number), I think they are:

1. PCC Reset (from the MCP).
2. Timer Interrupt (for time-critical tasks).
3. MCP. The MCP will use interrupt lines to signal the PCC to perform an action on the MCP’s behalf.
4. Scheduler.

Note that servicing an interrupt on the average requires 33 i960 processor cycles, or more than one cell cycle. In general, it will be preferable to poll for communication.

### 6.4 Initialization

The current PCC design has no boot ROM. Instead, the MCP will hold the PCC in reset, load boot software into the dual-port RAM, then release the PCC from reset to boot out of the dual-port. As of this writing, Jim was conferring with Dave Milton that VxWorks can run with such a bootstrap mechanism.

### 6.5 Interprocessor Messages

The optional VxWorks product VxMP contains support for shared-memory semaphores and message queues. It appears that these mechanisms require more memory than we would like; we are still evaluating whether to use VxMP or to write our own message package.

The initial software will send messages between MCP and PCC using an RPC model. The MCP may make one request at a time. The PCC will serve that request, then indicate to the MCP that it has completed the request. We will probably move to a more complex mechanism for later versions of the software.

### 6.5.1 Messages from MCP to PCC

Note: this section currently contains actions which might be performed by the MCP directly, if the MCP has direct access to the PCC bus.

1. Get Error Statistics
2. Setup VC input side
3. Setup VC output side phase 1
4. Setup VC output side phase 2
5. Teardown VC input side
6. Teardown VC output side

The Software Specification Document lists the following as direct access:

1. Receiver-Grant-Credit
2. Sender-Reply-Grant-Credit
3. Sender-Request-Credit
4. Sender-Reply-Request-Credit
5. Receiver-Query-Credit
6. Sender-Reply-Query-Credit
7. Receiver-Set-Time-to-Live-for-Credit
8. Sender-Reply-Set-Time-to-Live-for-Credit

I also think that we need to support the following:

1. Reprioritize VC
2. Set data, credit timeouts for a VC
3. Set bandwidth limit (X cells in Y time) for a VC.
4. Switch Modes to Run, Single-Step, Loopback, Test Signal Generator, Recovery, or Debug Monitor
5. Download new PCC software
6. Download new Scheduler software

### 6.5.2 Messages from PCC to MCP

1. VC has timed out credit
2. VC has timed out data
3. VC has exceeded bandwidth

### 6.5.3 Messages from PCC to Scheduler

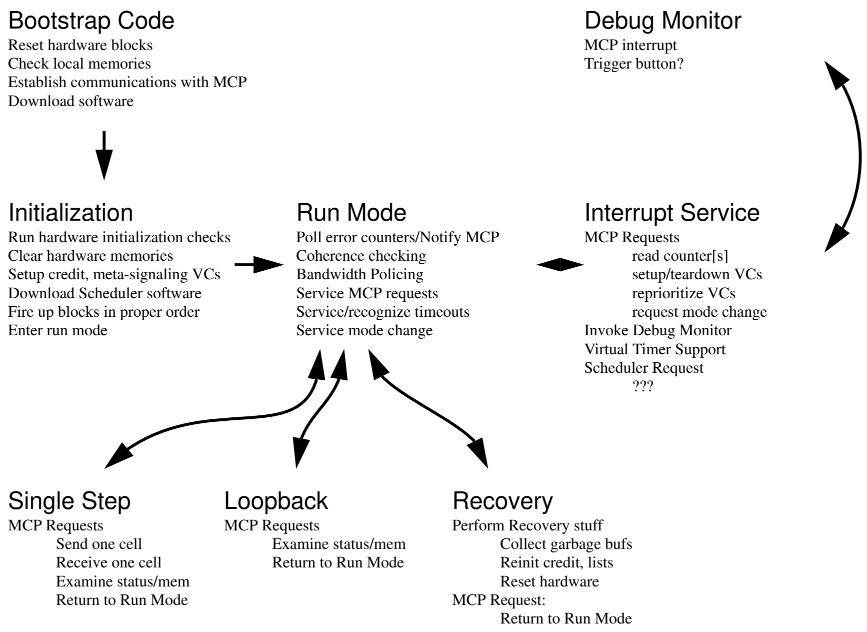
1. Download software
2. Enter single-step mode

**3.** Send one cell in single-step mode

### 6.5.4 Messages from Scheduler to PCC

I don't currently know of any.

## 6.6 Software Structure Diagram




Major Software Functions/Modes

## 6.7 Pseudocode

### 6.7.1 Bootstrap

The bootstrap routine is burned into the i960's boot EPROM (the Debug Monitor may also reside on the EPROM for system resilience). Ideally, it is a very short routine that we don't have to modify often.

Bootstrap Pseudocode:

Turn all blocks except other microprocessors off.
Test local memory.
Establish communications with the MCP.
Download rest of code from MCP.
Jump into initialization part of downloaded code.

### 6.7.2 Initialization Mode

Initialization is software downloaded from the switch workstation to the MCP to the PCC. As such, it is relatively easy to reconfigure. This stage includes all initialization of hardware blocks and all initialization of PCC data structures.

Downloaded Initialization Pseudocode

- Download programs to Scheduler, other microprocessors (if any).
- Test and initialize hardware blocks (routines supplied by block designers)
  - Test and initialize hardware block memories.
  - Set initial values of hardware block registers.
- Initialize PCC local data structures:
  - List of VCs on which to perform traffic/bandwidth policing.
  - List of VCs on which to perform credit/data aging
- Initialize PCC-MCP shared data structures:.
  - MCP->PCC command queue.
  - Polling cache for status registers on the port card.
- Start up blocks (exact order TBD):
  - CLM/VAR/ALM Pipeline
  - CTX, CRX
  - CCM and CCG
  - NTX
- Set up bootstrap VCs according to ATM standard (probably under MCP control):
  - Metasignaling VC
  - Credit VC[s] for this link.
- Wait for signal from MCP.
- Start Scheduler, NRX
- Enter Run, Single-Step, or Debug Mode

### 6.7.3 Run Mode

Continuous Loop through:

- Poll MCP connection for new commands:
  - Set up a VC.
  - Tear down a VC (complex: must drain LM data structures).
  - Modify an existing VC.
  - Request for Status Information.
- Perform Policing Functions
  - Figure out what time it is
  - Look at the VCs whose bandwidth-checking time has come
  - If they have exceeded their bandwidth, notify the MCP.
- Perform Coherence Checks
  - +++ What are these?
- Poll hardware block status registers.
- Perform Software Filtering functions (drain IVC FIFO)
- Enter Single-Step, Debug, or Breakpoint Mode, if requested.

### 6.7.4 Interrupt Service Routines

Hardware Reset Interrupt

This probably uses the i960's NMI (non-maskable interrupt) line.
Dump state to the MCP?
Enter debug monitor?
Warm start?

Timer Interrupts

Poll and reset whichever things need to be polled (e.g. bandwidth counters).
Set the time for the next interrupt.
Return.

MCP Interrupts

Figure out what the MCP wants the PCC to do.
If it's quick, then service it.
Otherwise, save the request in a command queue for processing by the main loop.

Scheduler Interrupts

Since I don't know what the Scheduler sends to the PCC, I don't currently have a list of things for this interrupt to do.

### 6.7.5 Debug Monitor

Read/Write a processor register
Read/Write a memory location
Set a breakpoint
Go
Single-Step
etc.

### 6.7.6 Single Step Mode

Put NRX, Scheduler into Single-Step Mode
Wait for system to stabilize
Loop through:
- Gather lots of statistics, status information.
- Wait for command from MCP:
  - Allow changes to status, programs (may require downloading more code).
  - Download statistics
  - Download entire state (core dump).
  - Send/Receive one cell.
  - Switch mode

### 6.7.7 Loopback Mode

Engage Loopback Paths
Loop through:
- Answer information requests from MCP
- Switch mode, if requested.

**6.7.8** Recovery Mode
Turn NRX, Scheduler off.
+++Reconcile this port card's data and control to itself.
Send inter-port-card information to MCP as clearinghouse.
Return to Run Mode on signal from MCP

# 7.0 Design Verification Plan

# 8.0 Test Plan

# 9.0 Delivery Schedule

# 10.0 Open Issues

This section will be empty in the final draft. It documents issues that have been identified but have not been resolved.

1. How does the PCC obtain timestamps? Probably can't count instruction cycles.
2. What kinds of electrical loopback paths will we implement?
3. What kind of single-step mode will the hardware support?