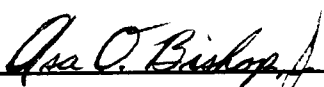


To the Graduate Council:

I am submitting herewith a thesis written by Bobby Ray Whitus entitled "Data Communications for a Large Signal Processing System." I have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

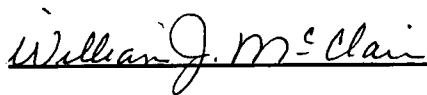


Asa O. Bishop, Jr., Major Professor

We have read this thesis
and recommend its acceptance:



Gerald Deegan CS



William J. McClain

Accepted for the Council:



Associate Vice Chancellor and
Dean of The Graduate School

DATA COMMUNICATIONS
FOR A
LARGE SIGNAL PROCESSING SYSTEM

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Bobby Ray Whitus
December 1996

Acknowledgments

I would like to extend my sincere thanks to Dr. Asa Bishop for providing support and encouragement to me both as an undergraduate and graduate student. He taught my first Electrical Engineering class and my first data communications class. Also, he has supported the completion of this thesis. Thanks to Dr. Tom Dunigan and Dr. William McClain for serving on my thesis committee. Dr. Dunigan's course in UNIX Network Programming provided the UNIX knowledge necessary for the completion of the work this thesis is based on.

Thanks to Jim Jansen and Randall Wetherington of the Oak Ridge National Laboratory's Instrumentation and Controls Division for the opportunity to work on the project that included the problem for this thesis. Thanks also to the other members of the project team that developed the measurement system described in this thesis.

Special thanks to my wife Cheryl for her patience, encouragement and help in completing this thesis and the course work necessary for a Masters Degree.

Last, but most important, I thank my Lord and Savior Jesus Christ for providing the opportunity, strength and ability to complete this process.

Abstract

Software for data communications in a large multi-node digital signal processing system (Beamformer) was designed and developed. The system hardware includes a high speed memory system that contains two vector processors, three Cray Research, Inc. (CRAY) Advanced Parallel Processors (APPs), and three Sun UNIX servers. High Performance Parallel Interface (HIPPI) links provide high speed communications paths between the APPs and Sun nodes. An Ethernet Local Area Network (LAN) is used for control message communications within the system and provides the links to external systems.

The research included development of client server software to allow access to Sun disks by APP programs via HIPPI. The research also included tests to measure the actual HIPPI data transfer performance. The results of these tests are presented.

A design using HIPPI and Ethernet was developed and implemented for the data communications within the Beamformer system and with external systems. The author developed a library of FORTRAN-callable C functions to access the HIPPI links from the CRAY APP programs and a key Sun program to receive spectral products from the APPs via the HIPPI interfaces. This program was tested and, along with the other communications code discussed, is working properly in the production environment.

Preface

A description of a system for underwater acoustic measurement (Measurement Facility) is given in Chapter 1 of this thesis. The description is given to supply the necessary background for the discussion of the data communications design and implementation for the Beamformer. The design and implementation of the complete Measurement Facility was the result of the efforts of a large team of scientists and engineers from several organizations.

Before the author became involved

- the top-level design for the Measurement Facility was developed;
- the Beamformer hardware had been specified, procured and delivered; and
- the signal processing design was underway.

The author became involved in the early stages of software development and integration.

The first problems assigned to the author were

- install, evaluate, and learn to use the Data Routing Process (DRP) software developed by another system contractor and
- develop client server software to allow APP programs to access Sun disk systems via the HIPPI hardware.

As software development continued, it became clear that an extensive communications design was needed. This led to the work described in Chapters 3 through 6. The DRP task is covered in Chapter 4, and the development of the HIPPI client-server software is described in Chapter 5. As part of the team effort, the design described in Chapter 6 was discussed with the team. After the data communications design was completed, the author developed the software described in Chapters 7 through 9.

Table of Contents

Chapter 1 System Description.....	1
Overview	1
Measurement Facility overview.....	2
Signal processing.....	2
Beamformer interfaces.....	6
Prior work	12
Chapter 2 Problem Statement	20
Introduction	20
The Problem	20
Requirements	20
Specific data transfer requirements	21
Objective and requirement drivers.....	21
Chapter 3 HIPPI Characterization.....	23
HIPPI Description.....	23
Performance Measurements.....	26
Chapter 4 Interprocess Communications Investigation	33
Introduction	33
Data Routing Process.....	33
IP Communications	35
Other UNIX IPC.....	36

TMS Memory System.....	38
Chapter 5 Client-server Software for HIPPI Access to Sun File Systems.....	40
Introduction	40
Characteristics of file IPC.....	40
Software development	42
Conclusions	50
Chapter 6 Design for Beamformer Data Communications.....	52
Introduction	52
Initial design	52
Measurements.....	61
Chapter 7 Library for APP HIPPI Communications.....	67
Introduction	67
Code description.....	67
APP data transmission.....	70
Chapter 8 Design of Software to Process APP Products	72
Introduction	72
Requirements	72
Original Design.....	72
Chapter 9 Details of the Process to Receive Spectral Products	83
Introduction	83
Interfaces and Specifications	83
Program architecture.....	86

Detailed program description	89
Conclusion	98
List of References	99
Appendices.....	101
Appendix A Listing of drp_src.c File.....	102
Appendix B Listing of drp_dest.c File	107
Appendix C Listing of open_ts_sun Function.....	110
Appendix D Listing of send_spls Function	112
Appendix E Listing of await Function.....	114
Appendix F Listing of rec_spls.h File	116
Appendix G Listing of spl_products.h File	121
Appendix H Listing of spl_rec_header.h File.....	124
Appendix I Listing of rec_spls.c File.....	126
Appendix J Listing of startup.c File	131
Appendix K Listing of rd_array.c File.....	137
Appendix L Listing of getargs.c File.....	140
Appendix M Listing of cshm.c File.....	143
Appendix N Listing of open_hippi.c File.....	147
Appendix O Listing of reset_hippi.c File	149
Appendix P Listing of hip_set_params.c File.....	151
Appendix Q Listing of process_pkt.c File.....	154
Appendix R Listing of merge_ap.c File	162

Appendix S Listing of wrt_fifos.c File	166
Appendix T Listing of snd_bips.c File	170
Appendix U Listing of snd_sotos.c File	173
Appendix V Listing of wrt_scnb.c File	176
Appendix W Listing of end_process.c File	180
Vita.....	183

List of Figures

Figure 1 Measurement Facility block diagram	3
Figure 2 Signal processing block diagram.....	7
Figure 3 Beamformer context diagram.....	8
Figure 4 Hardware block diagram.....	14
Figure 5 HIPPI Connections	24
Figure 6 HIPPI transfer sequence.....	25
Figure 7 Pseudocode for HIPPI test src programs	28
Figure 8 Pseudocode for HIPPI test dest programs.....	28
Figure 9 APP-to-APP Data Rates	31
Figure 10 APP-to Sun-Transfer Rates	32
Figure 11 Write using two commands	46
Figure 12 Read using two commands.....	47
Figure 13 Writes using <i>HipHostSkWrt()</i>	48
Figure 14 Reads using <i>HipHostSkRd()</i>	49
Figure 15 Initial design data flow diagram.....	55
Figure 16 Final design data flow diagram.....	62
Figure 17 Original APP0 HIPPI output timing.....	64
Figure 18 Final design APP0 HIPPI output timing.....	66
Figure 19 Basic APP async I/O flowchart	71

Figure 20 Original rec_spls context diagram.....	73
Figure 21 Original rec_spls pseudo code.....	76
Figure 22 Measured DRP write time	78
Figure 23 Measured disk write block time.....	79
Figure 24 Measured TCP write times	80
Figure 25 rec_spls context diagram	84
Figure 26 FIFO implementation.....	87
Figure 27 merge_ap data flow.....	94

List of Tables

Table 1 Measurement Facility component systems	3
Table 2 Beamformer context diagram data flows.....	9
Table 3 Sensor input data rates	11
Table 4 HIPPI test programs.....	27
Table 5 Measured HIPPI data rates.....	30
Table 6 Comparison of TCP and DRP data rates	35
Table 7 APP HIPPI file access client functions.....	44
Table 8 Processing functions for initial design.....	56
Table 9 HIPPI data channels.....	57
Table 10 DRP messages.....	57
Table 11 HIPPI port busy times.....	59
Table 12 New design HIPPI channels.....	63
Table 13 HIPPI port busy times for revised design	63
Table 14 Ashipio functions	68
Table 15 Original rec_spls terminator descriptions	74
Table 16 Original rec_pls data flow descriptions	74
Table 17 rec_spls terminator descriptions	85
Table 18 Data flow descriptions	85
Table 19 rec_spls command line options.....	87

Table 20 rec_spls source files	90
--------------------------------------	----

List of Acronyms

API	Application Programming Interface
APP	Advanced Parallel Processor
BDC	Beamformer Data Coupler
BF	Beamformer
BIP	Beam Image Product
Bps	Bytes per second
bps	bits per second
C50	SAM-2000 interface model
CNB	Coarse Narrow Band
CRAY	Cray Research, Inc.
DAC	Digital to Analog Converter
DMA	Direct Memory Access
DRP	Data Routing Process
DSP	Digital Signal Processing
FDDI	Fiber Distributed Data Interface
FIFO	First In First Out
FIP	Final Integration Product
FFT	Fast Fourier Transform
FPS	Floating Point Systems, Inc.
GB	Gigabyte

HIPPI	High Performance Parallel Interface
KBps	Kilobytes per second
LAN	Local Area Network
IP	Internet Protocol
IPC	Interprocess Communication
lpd	line printer daemon
MB	Megabyte
MBps	Megabytes per second
ms	milliseconds
NIS	Network Information System
NFS	Network File System
OTO	One-Third Octave
PE	Processing Element
PID	Process ID
PSI	Planning Systems Incorporated
SAM	Shared Attached Memory
SCNB	Short-term Integrated Course Narrow Band spectral product
SOTO	Short-term integrated OTO value
SPL	Sound Pressure Level
Sun	Sun Microsystems, Inc.
TLI	Transport Layer Interface
TMS	Texas Memory System

ULP

Upper Layer Protocol

Chapter 1 System Description

Overview

This thesis presents the design and implementation of data communications services for a large Digital Signal Processing (DSP) system. This DSP system is the Beamformer for an underwater sensor array. The goal of the communications services design was to provide the means to move control and real-time data between the Beamformer's internal subsystems and external data sources and sinks. The Beamformer and sensor array are part of a larger system that includes other measurement, data analysis and archival systems. For this thesis the phrase "Measurement Facility" will mean the Beamformer, sensor arrays and other systems to which the Beamformer interfaces.

This chapter describes the basic system, integration with other Measurement Facility systems and basic Beamformer signal processing. Also included in this chapter is prior research for the problem. Chapter two contains the data communications requirements. Later chapters describe the design, implementation and measured results.

Two complete Beamformer systems (legs) were built. Each leg receives data from an underwater array of more than 1000 hydrophones. The hydrophones are arranged in five groups. Each group is used to cover a specific frequency range and are sampled at different rates. Each of the five frequency ranges is called an aperture. Data communications was a key technology for the success of the Beamformer. Every

218 milliseconds (ms), one or more (depending on the aperture) 2048 16-bit data sample sets enter the Beamformer from each active sensor. This yields an aggregate input data rate of about 82 megabytes per second (MBps). Internal data rates as high as 31.5 MBps must also be supported.

Measurement Facility overview

The complete Measurement Facility consists of the sensor arrays, a telemetry system, two Beamformers, data display workstations, archival nodes, a target tracking system and a Digital to Analog Converter (DAC) system. The telemetry system includes analog to digital conversion, multiplexing, a fiber optic underwater-to-surface link and de-multiplexing components. Figure 1 contains a simplified block diagram of the total Measurement Facility. A short description of each component in Figure 1 is given in Table 1.

Signal processing

This section describes the basic beamforming method and calculation of output spectral products. More detailed documentation of some signal processing algorithms are also referenced.

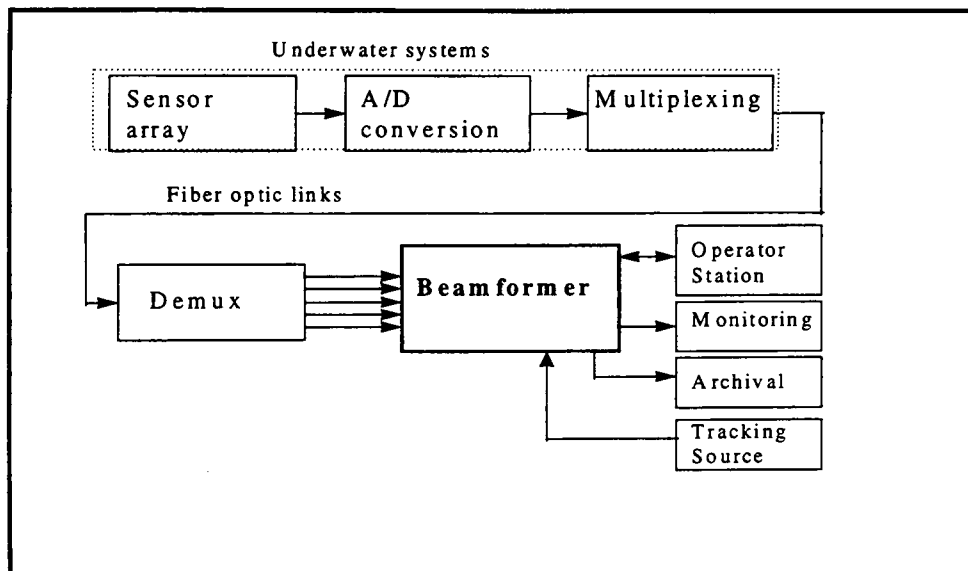


Figure 1 Measurement Facility block diagram

Table 1 Measurement Facility component systems

Component system	Description
Sensor array	Array of hydrophones arranged in five groups to cover five frequency ranges (apertures).
A/D conversion	Signal conditioning and analog to digital conversion.
Multiplexing	Electronics to multiplex digital sensor data. All data from each sub-array that covers one aperture is sent over a separate fiber.
Demux	The above water portion of the telemetry electronics that de-multiplexes the data from the fiber and provides a parallel interface to the Beamformer.
Beamformer (BF)	DSP system to generate beams from the digital signal data and produce several types of spectral products.
Operator Station	Primary operator station for setup and control of the entire system.
Monitoring	Third Octave Monitoring Station. Workstation used to view spectral products.
Archival	System used to archive data from Beamformer and other systems.
Tracking	Provides the Beamformer with target location information.

Beamforming is a technique that allows significant signal-to-noise improvements in measurements that are dominated by background noise. Beamforming is accomplished by phase-summing the output of a number of discrete omni-directional sensors into a single measurement with a directional response pattern (beam). Sensors are arranged in two cylindrical arrays. A separate Beamformer is used with each array, forming one system leg. Both continuous and non-continuous beams are formed. The continuous beams are used for accurate Sound Pressure Level (SPL) measurement and are output to a DAC as time domain data. The non-continuous beams are output as directed acoustic images. Twelve continuous and 80 non-continuous beams are produced.

Beam formation in this system is accomplished with a two-stage Fast Fourier Transform (FFT) beamformer. Sensor data is transformed into complex spectral values with an FFT. Next, a dot-product is used to form vertical beams. These vertical beams have a directional sensitivity in the vertical direction but a flat response in the horizontal direction. A second stage of beamforming forms the spot beams by applying another dot-product.

The coefficients for beamforming are generated from archived values that have been computed, verified and stored on disk. Files generated by the system operator specify beam characteristics that select the set of archived values. Real-time steering data is used at run time to generate coefficients from the archived values. Coefficient generation

consists of interpolating new coefficients from archived values based on the target position.

Sixteen-bit digital data for selected measurement beams is output to the DAC. This time series data is reconstructed from the frequency domain beam data. Details on the time series reconstruction can be found in the University of Tennessee Thesis "Reconstruction of a Composite Time Series Output for a Multiple-Aperture Frequency-Domain Beamformer" [Tucker, 1995]. The DAC time series output is used by other signal processing systems to perform specialized data analysis.

Power spectra are computed in the form of Coarse Narrow Band (CNB) SPLs for the continuous beams. The frequency domain outputs are Hanning weighted [Openheim, 1989] and the results summed over a 218 millisecond time period (frame) forming short-term integrations. The short-term integrations are stored for a measurement sequence called a run. The stored data are later used to produce Final Integration Products (FIPs). The final integration includes calibration. The system also supports re-integration of stored data until it is overwritten by the execution of another run.

Power spectra output is also produced as One-Third Octave (OTO) [ANSI, 1984] bands for all continuous (measurement) beams. OTOs are formed by summing the appropriate Hanning weighted power frequency bin values over one frame period. The OTO bands are combined from all apertures to yield a single set of values for each beam. For

apertures where multiple sample sets are processed each frame time, the OTO data is integrated forming Short term integrated OTO (SOTO) data. These SOTOs are output at the frame rate. Twenty-five real values are produced for each of the twelve beams. The 80 non-continuous beams are used to produce 348 values that form an image of the noise around the target. These 348 values are called Beam Image Products (BIPs). The BIP power values are produced in two forms: 25 standard OTO bands and 10 custom bands. The custom bands can include sums of frequency bins from any aperture. BIPs are output at the frame rate.

A block diagram of the signal processing is shown in Figure 2. Some stages are performed on a per-aperture basis, and others, such as time series reconstruction, use data from multiple apertures.

Beamformer interfaces

The beamformer interfaces with multiple subsystems using a variety of methods. A context diagram for the beamformer is shown in Figure 3 and descriptions of the external data flows are given in Table 2.

The sensor data uses a proprietary Texas Memory Systems (TMS) Shared Attached Memory (SAM)-2000 interface. The SAM-2000 interface is a model C50 and has a 64-bit wide data path. One C50 is used for each aperture. The last telemetry system

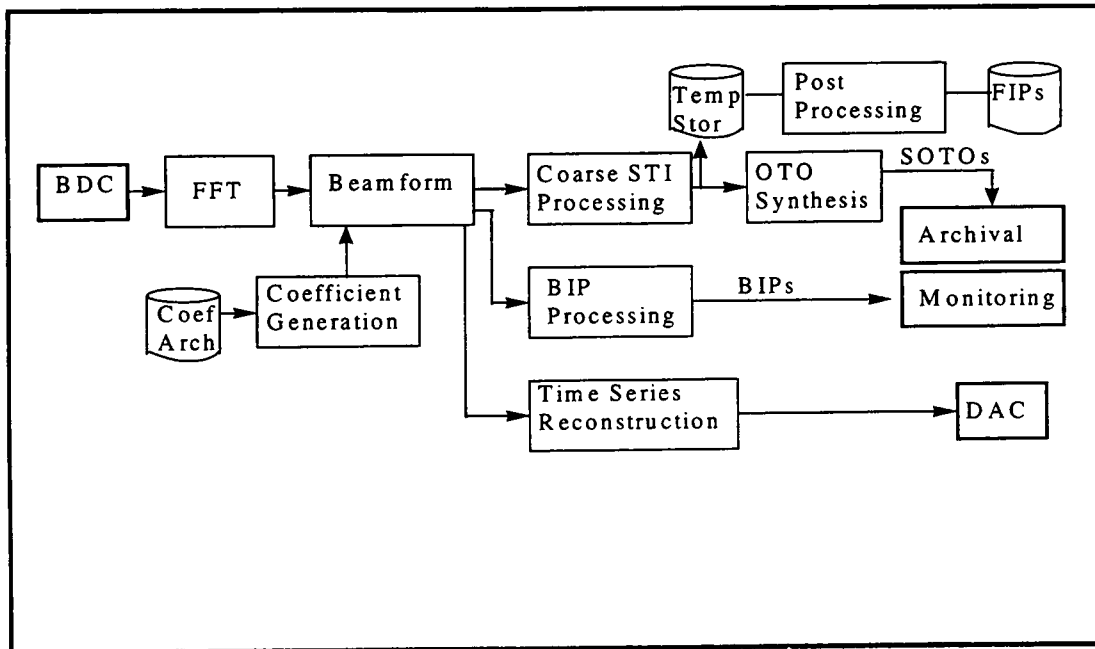


Figure 2 Signal processing block diagram

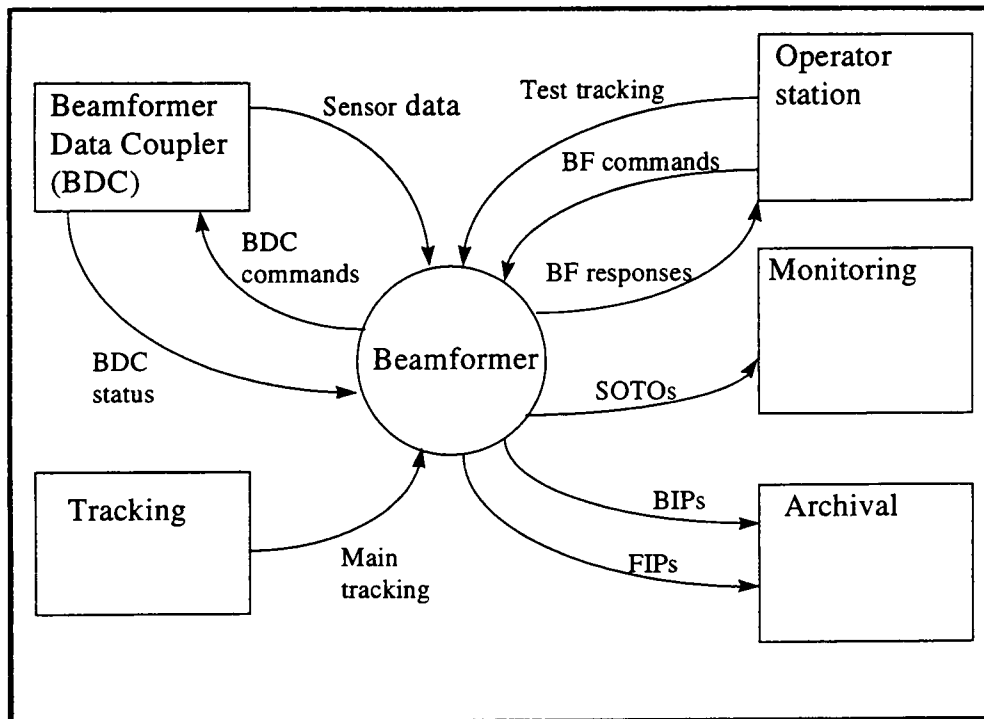


Figure 3 Beamformer context diagram

Table 2 Beamformer context diagram data flows

Data flow	Description
Sensor data	Digitized sensor data from the telemetry system. This is the principle input to the Beamformer.
BDC commands	Commands to control the BDC. Commands include starting the BDC in test or normal mode, stopping data flow and requests for status. Starting in test mode provides the beamformer with an input data stream from values stored in PROMS.
BDC status	Responses to state change commands and status values in response to status request commands.
Main tracking	ASCII messages specifying a beam steering direction. This steering information is derived from the target position. The nominal update rate is once per second.
Test tracking	ASCII messages specifying a beam steering direction. This steering is generated to measure ambient noise level with no target present. The nominal update rate is once per second.
BF commands	Commands to control the beamformer. Example commands include read setup data from shared disk files, start beamformer and perform calibration sequences.
BF responses	Error messages and responses to BF commands and status request messages.
SOTO	Short term integrated OTO values. These data packets consist of a header followed by 25 one third octave values for 12 measurement beams.
BIP	Beam Image Products. A set of 348 points per frame consisting of 25 one third octave values and 10 customs bands per point. The data packets consist of a header followed by the floating point data values.
FIP	Final Integration Products. This data consists of the results of data integrated over a large number of samples. The data is written to a Network File System (NFS) mounted disk at the completion of each integration as specified by the external systems.

component, the Beamformer Data Coupler (BDC), was designed to interface with the C50. The data rates are given in Table 3.

The BDC commands and responses use an RS232 physical interface . This asynchronous serial link operates at 9600 bits per second (bps). The average data rate for this link is on the order of 10 Bytes per second (Bps).

Asynchronous serial lines are also used to receive test tracking and main tracking data. These tracking sources send a short message once per second. The average data rate for these links is less than 100 Bps.

BF commands, BF responses, and SOTO and BIP data are passed over Ethernet using the DRP software. DRP is a software package written by Planning Systems Incorporated (PSI) that uses TCP/IP. Part of this research was the characterization of DRP which is described in Chapter 4.

BIPs and SOTOs are produced in real-time with average data rates of 224,256 Bps and 7021 Bps, respectively. The average data rates for BF commands and responses are less than 100 Bps.

Table 3 Sensor input data rates

Sensor data input (aperture number)	Data rate (MBps)
1	5.24
2	10.4
3	11.8
4	23.6
5	31.5

FIP results are written to file systems on external nodes using NFS [Stern, 1991] disk access. The data rate for this transfer is the maximum rate allowed by Ethernet and NFS which can approach 10 megabits per second.

Prior work

The beamformer hardware and several pieces of key software were purchased from Cray Superservers, a division of Cray Research, Inc. This CRAY division was formed when Floating Point Systems, Inc. (FPS) was purchased. The parallel processors used in the system were FPS products. In addition to the FPS parallel processors, the system contains hardware from several other vendors including UNIX servers from Sun Microsystems, Inc. (Sun). Two complete systems, referred to as legs A and B, were purchased.

The beamformer as supplied by CRAY was required to meet the following requirements.

- Perform FFT beamforming on test data.
- Process input data at the sensor data rate.
- Simulate output of all data products.
- Demonstrate computation bandwidth required for generation of all spectral products.

While the system was procured as a beamformer, significant additional software development was required before integration with the other Measurement Facility systems. The acceptance tests were designed to prove that the hardware and

beamforming software could meet design requirements but did not test end-to-end functionality. For example, the coefficients used to generate beams were not steered by external data, instead they were static values pre-computed on another system and read from a file. Also, output data streams were tested, but actual spectral products were not produced. No BIPs were generated by the delivered system; only sample calculations were performed to exercise the processors. The CRAY software that was used in the final system as delivered included optimized parallel processor code to perform FFTs and dot products. Some code from the input subsystem that reads in sensor data was also retained in the final system after modification. A block diagram of the hardware is shown in Figure 4.

The CRAY APP is a multiple-bus, parallel processor designed for matrix-oriented applications. All signal processing including coefficient generation, beamforming, time series reconstruction and SPL generation takes place in the APPs. There are three CRAY APPs in each Beamformer: APP0, APP1, and APP2. APP0 beamforms apertures one and two, generates coefficients for all apertures and performs time series reconstruction. APP1 beamforms apertures three and four, and APP2 beamforms aperture five. Each APP has a bit-width of 64, the capacity to hold 256 Megabytes (MB) of memory, a maximum of 85 Intel i860 processors, up to two HIPPI and a VME control interface. APP0 has 64 MB of memory, and APP1 and APP2 have 32 MB of memory each. All Beamformer APPs contain the maximum of 85 processors and two HIPPI interfaces. Eighty-four i860 processing elements (PEs) are arranged on seven buses, and one i860

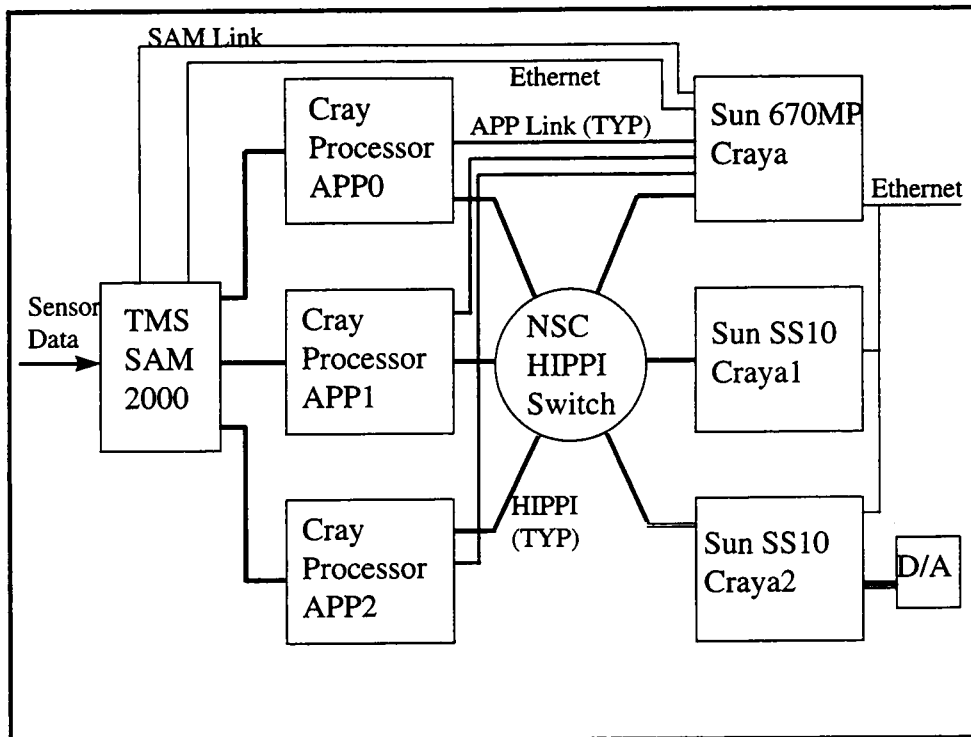


Figure 4 Hardware block diagram

is used as a control processor. The buses are connected to common memory through a cross-bar switch. Each APP operates under control of the Sun 670MP. A VME interface in the Sun 670MP provides the physical connection to the APPs with each APP having a separate VME controller. APP programs are compiled and linked on the Sun 670MP then downloaded over the VME interface for execution. The APP HIPPI channels can perform I/O at rates near 100 MBps. Both HIPPI interfaces share one port of the cross-bar switch limiting the total I/O rate to 160 MBps. HIPPI channel zero from each APP connects to the HIPPI switch. HIPPI channel one on each APP connects directly to a separate HIPPI-SAM interface.

The TMS SAM-2000 is a large, multi-ported memory system. Its main purpose is to manage and manipulate the incoming sensor data so that each CRAY APP receives the appropriate data in the correct format. In addition to providing the sensor data input path, a section of SAM-2000 memory is used for flags and control data. This common memory area is used to control and synchronize data acquisition and APP program execution. Sun node Craya accesses the common area to start and monitor data acquisition. Four types of modules are used in the SAM-2000: Super Scientific Processor (SSP-160), memory, C50 parallel interface and Verification Processor (VP32). The SAM-2000 is configured with 256 MB of memory and has the capacity to hold 4 MB of memory. Two SSP-160 cards are used to preprocess the sensor data. Each SSP-160 provides up to 160 million floating point operations per second of processing capability. Programs for the SSP-160s are developed and downloaded to them from the Sun 670MP via an Ethernet interface. Nine

C50 interface cards are contained within the SAM-2000. The C50 cards can be configured for 16, 32 or 64-bit wide data paths. Five C50 interface cards are used for input of sensor data (one for each aperture), three C50s for output of sensor data to the appropriate CRAY APP and one C50 for a control interface back to the Sun 670MP. The C50s connected to the HIPPI-SAM interfaces and the BDC are configured for a bit-width of 64. The C50 connected to the Sun 670MP SBus interface is configured for 16-bit transfers. The VP32 serves as a system controller, is used to run diagnostics and maintains a log of memory errors. VP32 diagnostics are executed, and the error log is accessed via the VP32 Ethernet connection.

Three separate Ethernet connections are used in the TMS SAM-2000, two connections for the SSP-160 and one for the VP32. The only other connection to this Ethernet segment is the Sun 670MP. A multi-port Ethernet transceiver is used to connect the three Sam Ethernet ports and a Sun 670MP port together. This forms an isolated Ethernet segment. Three HIPPI-SAM units provide HIPPI interfaces to the TMS SAM-2000 memory system. These interfaces allow read/write access to the memory system from HIPPI channels. The HIPPI-SAM units interface to the SAM-2000 via a 64-bit TMS C50 interface. Craya manages the SAM-2000 via the SAM-2000's SB-50/C50 parallel connection and the private Ethernet. The SB-50/C50 connection is used to read and write to a section of SAM-2000 memory that contains flags and status information. The Ethernet connection is used to load the SSP-160 programs and to communicate with the VP32 card. TMS supplied libraries are used to support Sun 670MP to SAM-2000

communications. A proprietary protocol is used to access SAM-2000 memory from HIPPI. The CRAY APP software includes a driver that allows the user to seek, read, and write to SAM-2000 memory.

Craya is a Sun 670MP dual processor server with 64 MB of memory. This node executes the main Beamformer control task and controls execution of APP and TMS SAM-2000 SSP-160 programs. The Sun 670MP is also used to develop the SSP-160 and APP executables and runs the Sun Network Information System (NIS) [Stern, 1991] service. Both VME and SBus adapters are natively supported for I/O. Three VME slots are currently used by the APP interfaces. Three SBus adapters are used: an Ethernet/SCSI adapter, a TMS SAM-2000 interface, and a KSI TPRO-S/tSAT-S synchronizable time generator. The system has an internal SCSI channel that does not use SBus or VME bus slots to connect internal devices. Internal SCSI devices include a 2.3 Gigabyte (GB) tape drive, a 0.25 inch tape drive (150 MB) and two 1.2 GB hard disks. The only external SCSI devices connected to the SBus adapter is one 1.2 GB disk drive.

Craya1 is a standard Sun SPARCstation10 with a Ross hyperSPARC processor upgrade. Craya1 is used to receive spectral products and forward them to their destination, receive steering data and send it to APP0, perform the final integration processing, and control the BDC. Spectral product destinations include both internal disk and external systems (via Ethernet). Final integration results are written to a file system mounted from a remote system. This node contains one 90 Megahertz hyperSPARC processor and a 64

MB main memory. Three of four SBus slots, which provide I/O expansion, are used.

One SBus slot is used by a serial port expander that has eight RS232 connections.

Another SBus slot is occupied by an unused S11W parallel interface. The third SBus slot is used by a HIPPI interface. This node also has a SCSI channel that connects to two internal disk drives. Internal disk storage capacities are 400 MB and 1 GB.

Craya2 is a Sun SPARCstation 10 Model 30 with a single 33 Megahertz processor. This node is responsible for sending time series data to the DAC. Four SBus slots provide I/O expansion. One SBus slot is used by a HIPPI interface, and two others are occupied by S11W 16-bit parallel interfaces. The fourth is not used. The S11Ws emulate the Digital Equipment Corporation DRV11 parallel interface. This node also has a SCSI channel that connects to two internal disk drives. Internal disk storage capacities are 400 MB and 1 GB.

Two Sun SS5 workstations are used in the systems. Craycon is used as the console device for both Beamformers, and Craydevel is used for software development. The hardware is identical on both systems so Craydevel can serve as a backup for Craycon. The Sun SS5s are single processor SPARCstation 5 workstations with a 32 MB memory. Three SBus slots provide I/O expansion on each workstation. On both, one of the SBus slots is occupied by an eight-port serial adapter and another by the video frame buffer card. Craydevel also has a printer card. The third slot on Craycon is not used. Disk storage includes an internal 400 MB drive and an external 1GB drive. Three of the SBus

serial adapters's ports on Craycon connect to the console ports on the leg-A Beamformer, and three connect to the leg-B Beamformer.

The Network Systems, Corp. PS8-8 HIPPI Switch allows communications between any of the connected devices. The switch, in effect, converts a group of point-to-point connections into a local area network. Each node has two 32-bit connections to the switch, a source and destination. When a source wishes to send a packet to a destination, it programs the destination address into the HIPPI packet I-Field. Two options are available to determine how a busy destination is handled. A busy destination means that another input port is connected to that destination. If the "camp on" bit is set in the I-Field of the source packet, the source will wait on the destination to become free. If that bit is not set, a "connection rejected" signal will be sent to the source which causes the write to fail. Six ports of the eight-port switch are used with the Beamformer.

Chapter 2 Problem Statement

Introduction

This chapter states the requirements for the Beamformer communications design and implementation. First, the problem is given; next, the general requirements are listed; then the specific data transfer requirements are listed. The requirement drivers are discussed in the last section.

The Problem

The goal of the Beamformer data communications task is to design software to perform all necessary data transfers in an efficient manner. The objectives also include providing simple interfaces for developers of Beamformer signal processing code and ensuring that all real-time data is delivered on time.

Requirements

1. APP-to-APP and APP-to-Sun transfers must use HIPPI links.
2. APP code must be in the form of FORTRAN-callable C language functions.
3. The Sun nodes must run the SunOS operating system.
4. File I/O for the purpose of interprocess and interprocessor communications must be given priority.
5. All communications with other Measurement Facility systems must use NFS disk files or the DRP.

Specific data transfer requirements

Beamforming for apertures one and two, coefficient generation and time series reconstruction, must run on APP0. Beamforming for apertures three and four must run on APP1. Only beamforming for aperture five may run on APP2. Coefficients generated on APP0 must be transmitted to APP1 and APP2 every frame time. Time series data for reconstruction must be sent from APP1 and APP2 to APP0 every frame time.

Objective and requirement drivers

The first objective listed is to promote modular development of code. Separating communications from the signal processing code allows parallel code development. This also improves code modularity and maintainability.

The second objective is fundamental to the system. If any of the real-time data transfers do not occur on time, system synchronization is lost, and the system must be restarted.

The first requirement is a result of the way CRAY integrated the hardware. Another option for APP-to-APP and APP-to-Sun transfers would be to use the TMS SAM-2000 memory system for all transfers. However, the SAM-2000 can not be configured with enough I/O cards to provide the number of Sun interfaces required to implement this option.

The second requirement is based on the fact that signal processing code is written in FORTRAN while it is necessary to perform I/O in C.

The third requirement is to provide a common operating system on all Beamformer Sun nodes. Another operating system option is Sun Solaris. Solaris could not be used on Craya because the CRAY APPs only work with SunOS. Solaris provides better real-time performance, but mixing operating systems in the Beamformers would increase system administration costs.

The fourth requirement is based on advantages of file Interprocess Communication (IPC) as listed in Chapter 5.

The fifth requirement results from interface agreements between Oak Ridge National Laboratory and another Measurement Facility system provider.

Chapter 3 HIPPI Characterization

HIPPI Description

HIPPI links provide high speed communications for the Beamformer. HIPPI interfaces can provide 800 megabits per second or 1600 megabits per second communications paths over 32 or 64-bit parallel data paths. Each HIPPI link provides a simplex link between a source and destination. Full duplex connections between two nodes is implemented by using two links. HIPPI switches (hubs) may be used to interconnect multiple nodes creating a LAN. The switches connect a source node to a destination based on an address specified at the start of a data transfer.

The HIPPI physical layer protocol provides transfer of data packets from a source to a destination with or without a switch in the path. Upper Layer Protocols (ULPs) are defined for use over the physical layer. Defined protocols include Internet Protocol (IP) and memory access. ANSI standards X3.183-1991 [ANSI, 1991], S3.210-1992 [ANSI, 1992] and S3.222-1993 [ANSI, 1993] define HIPPI. The HIPPI physical connection consists of 100 wire cables carrying 44 signals as shown in Figure 5.

In addition to the 100 wire copper standard, a version of HIPPI that uses serial transmission over fiber has been developed. However, there are currently no plans to submit this implementation for standardization by ANSI. The Beamformer uses only copper connections.

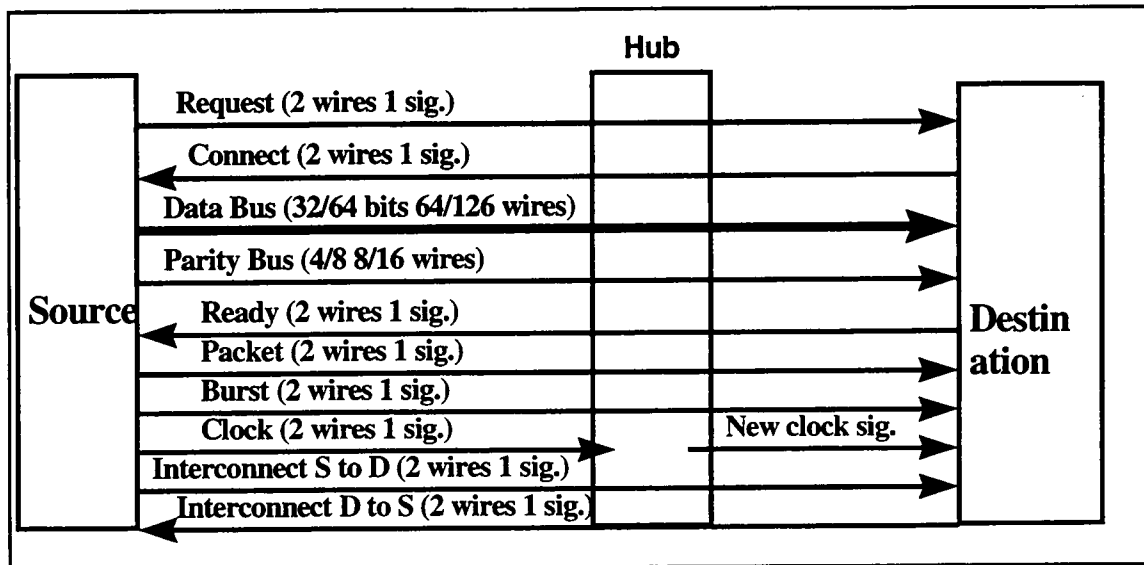


Figure 5 HIPPI Connections

Data transfers consist of a connection sequence followed by the data transfer and a disconnect. This sequence is shown in Figure 6. During the connection sequence, a destination address is issued to command a switch to pass the connection to the correct output port. If a switch output port is in use, the connection cannot be established. The source may wait on the port to become available by using the “camp-on” bit in the command. If the “camp-on” operation is not set by the source node, it will receive a connections rejected indicator. If a direct connection between source and destination nodes is made without a switch, the address and “camp-on” bits have no effect.

The HIPPI devices may be used in raw mode or with the framing protocol. In raw mode, unformatted byte streams are passed between devices. When raw mode is used, only one HIPPI file descriptor on a node can be opened. This requires all communications to be passed through the one connection. The framing protocol places a header on each data

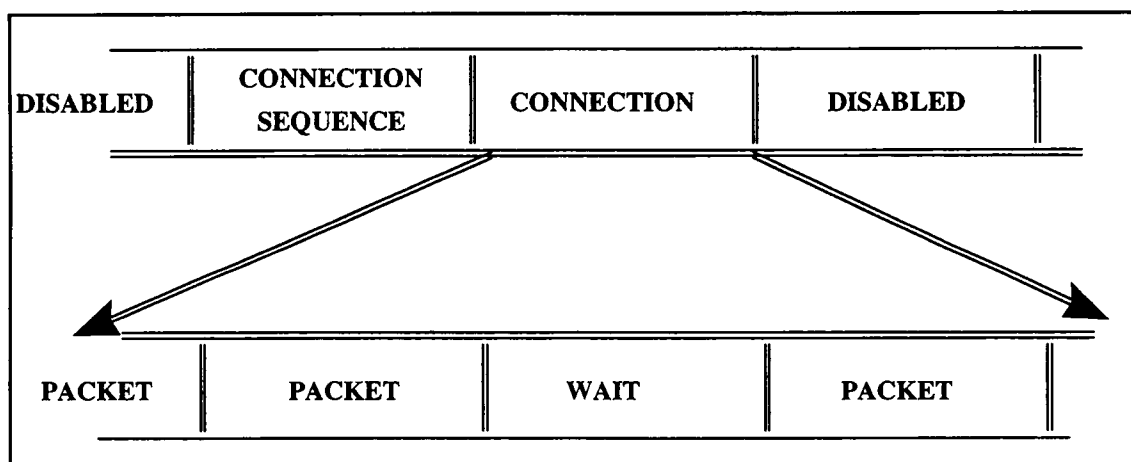


Figure 6 HIPPI transfer sequence

transfer that specifies a user or standard upper layer protocol. As stated earlier, the upper layer protocol may be IP, user defined, or memory access. Using the framing protocol allows multiple processes to open multiple file descriptors. This allows multiple data streams to be established across one physical link. Framing protocol is used for all APP-to-APP and APP-to-Sun data paths. ULPs are defined for the different logical data paths.

The device drivers for the CRAY APPs and Sun VME interfaces provide raw and framing protocol services. No higher level protocols are supported except for proprietary TMS SAM-2000 access from the APPs via HIPPI-SAM interface units.

Performance Measurements

Three performance tests were run to characterize the HIPPI links, measuring the maximum data rates for varying size transfers. The test programs are listed in Table 4. All of the source programs send a simple numeric ramp as the data packet. The destination may be used to check data validity by specifying a command line switch, but no data checking is done for the performance test since this would slow the receiving process.

Pseudocode for the source (data sender) and destination (data receiver) programs are shown in Figure 7 and Figure 8, respectively. The code for the Suns and CRAY APPs is very similar, differing only in device specific system interfaces.

Table 4 HIPPI test programs

Program	Description
app_src	APP program to send packets to a destination node and measure the data rate. Transfer length, number of packets, destination node, mode (raw or framing protocol) and a file to which results are recorded may be specified as command line arguments.
app_dest	APP program to receive data packets. Transfer length, number of packets, destination node, mode (raw or framing protocol) and a flag specifying that the data values in the packets be verified may be specified as command line arguments.
sun_src	Sun program to send packets to a destination node and measure the data rate. Transfer length, number of packets, destination node, mode (raw or framing protocol) and a file to which results are recorded may be specified as command line arguments.
sun_dest	Sun program to receive data packets. Transfer length, number of packets, destination node, mode (raw or framing protocol) and a flag specifying that the data values in the packets be verified may be specified as command line arguments.

Pseudocode for "src" programs

```
main
{
    read command line arguments: destination, length_of_transfers, mode
    (raw or framing_protocol), number_of_packets and results_file ;
    open hippi device;
    set HIPPI ifield t destination;
    total_time = 0.0;
    for (number_of_packets)
    {
        stime = time_from system;
        write packet;
        total_time = total_time + (time_from_system - stime);
    }
    calculate data_rate;
    print length and data_rate;
    If (results_file)
    {
        write results to results_file;
    }
}
```

Figure 7 Pseudocode for HIPPI test src programs

Pseudocode for "dest" programs

```
main
{
    read command line arguments: destination, length_of_transfers, mode
    (raw or framing_protocol), number_of_packets and verify_flag ;
    open hippi device;
    for (number_of_packets)
    {
        read packet;
        if(verify_flag)
        {
            if (values in received packet in error)
            {
                print error message;
            }
        }
    }
}
```

Figure 8 Pseudocode for HIPPI test dest programs

A number of different tests were run, because APP0 has a different amount of memory than APP1 and APP2 and different model Suns were used. APP0 is configured with 64 MB of memory while APP1 and APP2 have 32 MB of memory. Sun node Baby is a Sun 670MP with two SPARC processors. The 670MP includes a standard VME bus interface, so no external Sbus-to-VME adapter is needed. Table 5 contains the results of the data transfer rate tests. Data for APP-to-APP tests are graphed in Figure 9, and data for APP to Sun tests are graphed in Figure 10.

The test results show that the APP-to-APP data rates are fairly independent of memory size and protocol (raw vs. FP). However, the rates are very dependent on the size of the transfer. For the APP-to-Sun transfers, raw mode transfers yield higher data rates for the smaller transfer sizes, but the rates for the larger raw and FP transfers are close. The transfers rates are lower when using the Sun 670MP, because the 670MP's VME controller does not support block mode Direct Memory Access (DMA). The other Sun nodes use an Sbus to VME adapter that was modified to support block mode transfers.

Table 5 Measured HIPPI data rates

Length (Bytes)	Data Rates Between Nodes (KBps)							
	APP0 to APP1 FP	APP0 to APP1 RAW	APP1 to APP2 FP	APP1 to APP2 RAW	APP0 to Craya1 FP	APP0 to Craya2 FP	APP0 to Craya2 RAW	APP0 to Baby FP
16	10	10	10	11	3	3	11	7
64	42	30	39	44	11	12	45	27
256	149	118	148	146	32	38	157	117
1024	594	378	576	614	125	146	631	492
4096	2144	2416	1383	2337	350	484	1805	1180
16384	7919	5565	8249	8668	1502	1631	4859	2971
56536	22878	18955	22840	25480	4609	3973	8042	3654
262144	40951	55941	52896	56977	9219	7976	9642	5750
1048576	71485	71245	70949	77753	10628	9502	10142	6008
4194304	79042	81610	77744	81396	11155	9941	10247	6098
16777216	80743	81188	80823	82120	-----	Not measured	-----	
Legend APP0 Cray APP with 64 MB main memory APP0 Cray APP with 32 MB main memory APP2 Cray APP with 32 MB main memory Craya1 Sun SS10 with hyperSPARC processor Craya2 Sun SS10 with SuperSPARC processor Baby Sun 670MP with dual SPARC processors Raw Raw mode transfers FP Framing protocol used with user specified protocol number								

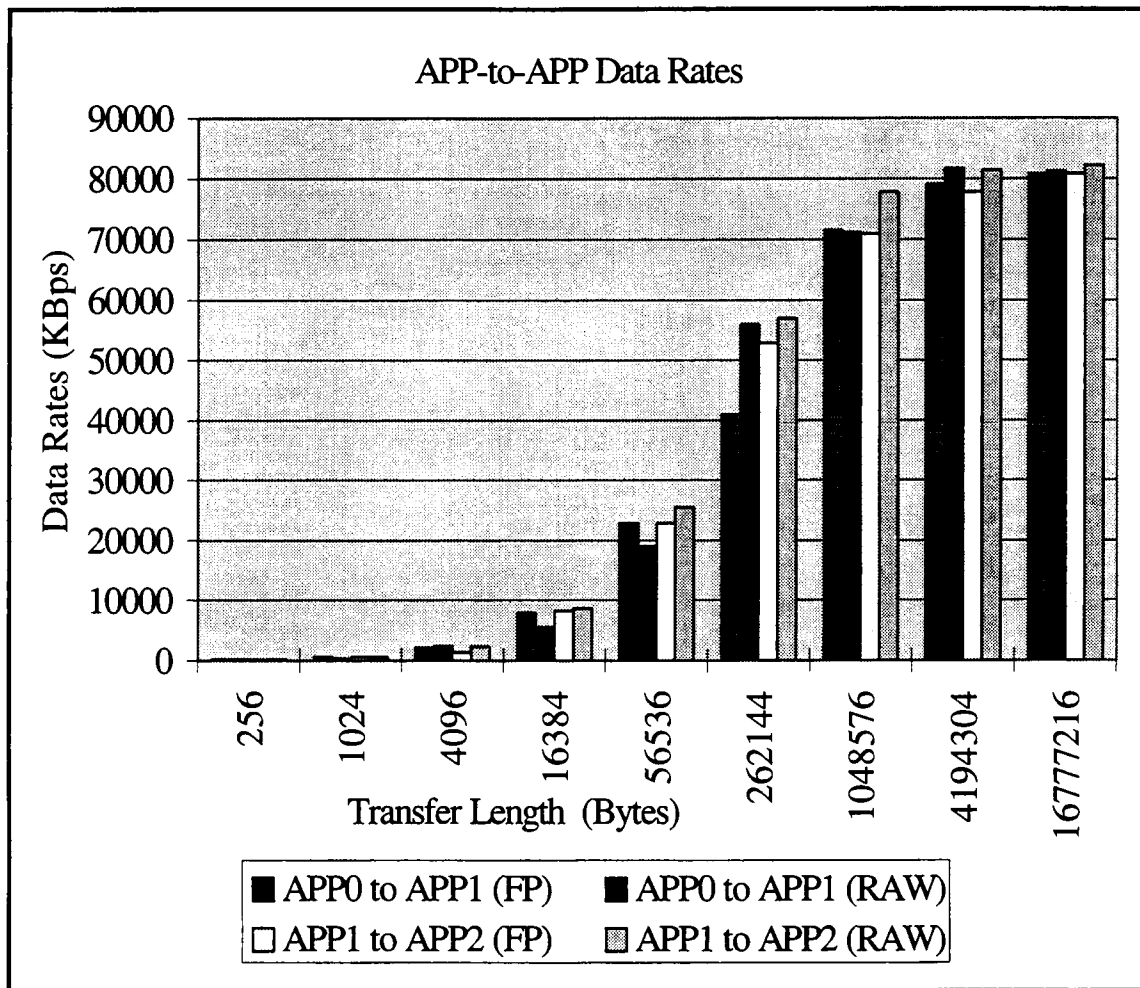


Figure 9 APP-to-APP Data Rates

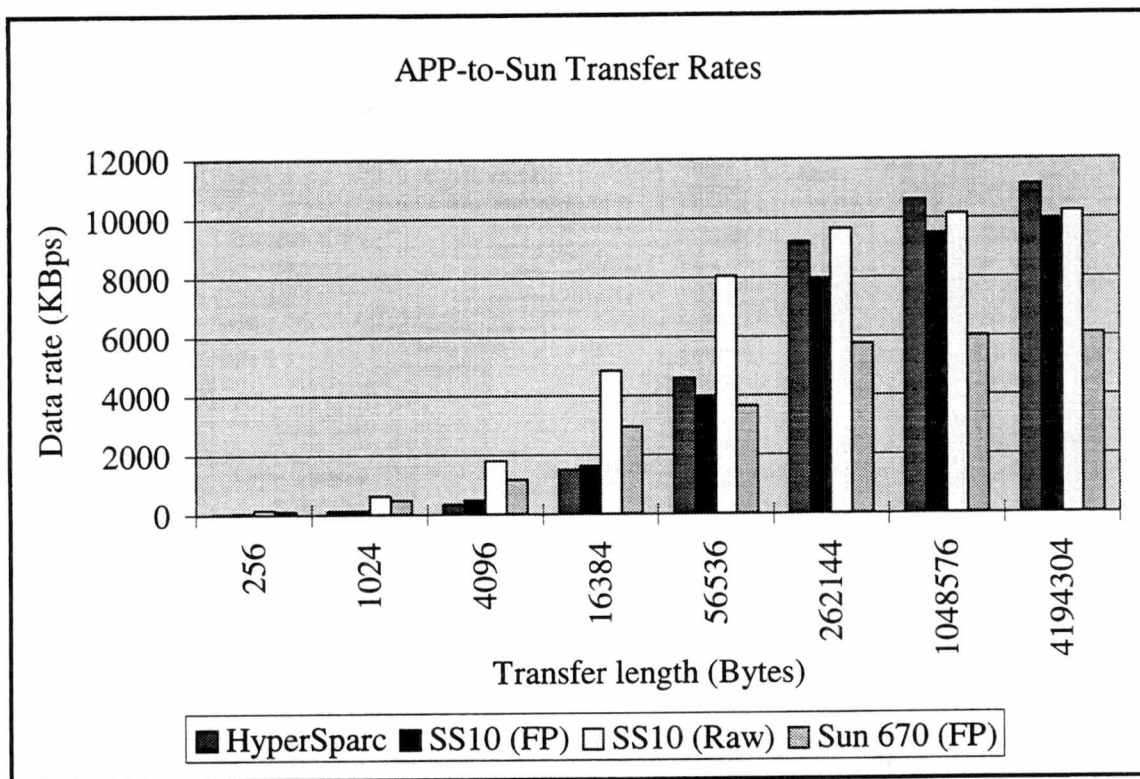


Figure 10 APP-to Sun-Transfer Rates

Chapter 4 Interprocess Communications Investigation

Introduction

In addition to the HIPPI channels, several other facilities are available in the Beamformer systems for interprocess and interprocessor communications. These facilities include DRP, standard UNIX Internet protocols [Stevens, 1990], UNIX IPCs [Stevens, 1992], and the TMS SAM-2000 memory system. All of these facilities were researched to determine which ones could be used to solve the problem. Also, the problem required the use of DRP for communications with external systems. More information is given on DRP in this chapter than for the other facilities, because no public documentation is available on this facility.

Data Routing Process

DRP refers to a software suite developed by PSI for message exchange between processes and processors. TCP, UDP, and IP are the underlying protocols used by DRP. DRP provides the user with a simple message-passing service that is much easier to use than Berkeley sockets [Stevens, 1990]. This suite of software includes the DRP process and the TCPIO library of functions that are linked to the user's code. This software runs on SunOS, Solaris, HPUNIX, and VMS systems.

Messages are passed from senders to one or more receivers. A process uses DRP to send a message by connecting to a DRP process, sending packets with a message type

specified and disconnecting when done. The DRP process that a user process connects to may be running on the same node or a remote node. Processes receive desired messages by specifying the message types in the connection to DRP. The receive function then returns messages of the type desired.

It is not necessary for senders or receivers to know node information; they simply send and receive messages based on message type. For example, one process may send a message of type1, and all processes on that or any other node on the LAN connected to type1 message would receive it. To the user this almost appears to be a broadcast system, but the messages are actually sent over TCP stream connections that are established by the connects. In the previous example, if one node has a process sending type1 messages and two processes on other nodes have DRP connections to receive type1 messages, two TCP streams would be established. The DRP processes learn about each other when DRP is started by sending broadcasts that are received by all other DRP processes on the LAN. TCP connections are then established as needed. A single process can perform both sends and receives.

Two simple programs used to measure data rates are given to provide an example of how DRP connections are used. Code in the source file, `drp_src.c`, listed in Appendix A, connects to DRP and sends messages. The code in the source file, `drp_dest.c`, listed in Appendix B, receives DRP messages.

During DRP characterization, tests were run to compare the data rate performance of DRP to simple socket connections. The results of the tests are shown in Table 6. The data shows that the performance penalty using DRP instead of TCP socket connections is moderate for large transfers but significant for small transfers. Based on the data rate for eight-byte transfers, approximately 125 ms is the minimum time required for each message transmission.

IP Communications

Berkeley sockets are a standard Application Programming Interface (API) [Stevens, 1990] for using TCP and UDP. Both of the protocols use IP. TCP provides a reliable stream between a client and a server. UDP sends data packets from a client to a server without guarantee of delivery. When using UDP, the sender of the packet has no knowledge as to whether the packet was received at the destination. TCP provides acknowledgments of received packets, and lost or corrupted packets are resent. The

Table 6 Comparison of TCP and DRP data rates

Transfer size (bytes)	TCP data rate (KBps)	DRP data rate(KBps)
8	89.8	0.1
32	37.8	0.3
128	140.1	1.3
512	387.3	5.1
2048	561.1	20.5
8192	630.8	82.4
32768	359.8	316.0
131072	579.7	341.5.4

reliability of TCP comes at the price of increased overhead. In order for a client to contact a server using TCP or UDP, the server address and service port number must be known. UDP also supports broadcast and multicast where all or a group of end nodes can receive a common message. TCP and UCP may be used for interprocess communications on the same node, in which case the messages are not sent over the LAN interface. UDP multicast was not considered for use with the Beamformer, because it is not supported in the standard SunOS kernel.

SunOS also includes the Transport Layer Interface (TLI) to TCP and UDP. TLI is not as protocol specific as the Berkeley sockets, making it useful with multiple underlying protocols. However, it is not as popular either. Since developers are more familiar with Berkeley Sockets APIs, the TLI APIs were not considered for Beamformer use.

For all the Beamformer nodes that use the IP protocol, the physical layer is provided by Ethernet. One Ethernet segment connects all Sun nodes. Craya has an additional private segment for connection to the TMS SAM-2000 system. An upgrade path for the physical layer is Fiber Distributed Data Interface (FDDI). FDDI is used by several of the Measurement Facility nodes but is presently not required by the Beamformer.

Other UNIX IPC

Pipes provide a one-way data stream connection. They are used for communication between parent-child processes or between processes that have a common parent. Pipes

are created with the *pipe()* function. Two file descriptors are returned, one for reading and one for writing. When pipes are used for parent-child communication, the parent executes *pipe()* before forking the child. The parent then must clone either the read or write file descriptor, and the child must close the other unused file descriptor.

Communication can be established by creating two pipes.

FIFOs (also called named pipes) are similar to UNIX pipes but have names associated with them to allow communication between unrelated processes. The *mknod()* function is used to create FIFOs. Once the FIFOs are created, they are accessed using standard I/O functions including *open()* and *close()*. As is the case for pipes, FIFOs operate as streams.

Message queues are part of the family of UNIX SystemV IPC. They are also supported under Berkeley-based SunOS. Message queues provide a mechanism for transferring fixed-length messages with a type-id between unrelated processes. The queues are maintained in the kernel, and data in the queues are persistent across process creation and termination. Message queues are bi-directional; messages may be written or read from the same queue. A message type is specified with each message written. Processes can read messages of a specific type or range of types. The size limit on message queues made this IPC facility unusable for transferring large data sets. By default, SunOS limits the number of bytes in message queues to 8192.

Semaphores are another SystemV IPC used for synchronization. The main use of semaphores is to synchronize resources shared by unrelated processes. The name comes from semaphore-signaling systems that prevent a train from entering a tunnel when another train is inside. The value of a semaphore is the number of resource units available. Functions are provided to test and change the value of a semaphore in one system call. If the value could not be tested and changed as an atomic operation, one process could alter the value between the time another process tested and altered it. Like message queues, semaphores are maintained in the kernel and persist across process creation and termination.

SystemV shared-memory functions allow multiple processes to have direct access to a common memory segment. This is the fastest form of IPC. When multiple processes use a shared memory region, some form of synchronization must be used. Semaphores are often used with shared memory for synchronization. While care must be taken to insure that data is not corrupted when using shared memory, it is very useful for creation of FIFOs and buffers between data sources and receivers.

TMS Memory System

The TMS SAM-2000 memory can be accessed by all the APPs and the Sun 670MP. The design of the SAM-2000 did not allow for interface to the other Sun nodes. The system was designed to use the SAM-2000-to-APP links as the data input path and to use a section of memory as a flag area. The flag area contains synchronization and status

information which can be accessed by the Sun 670MP. If the SAM-2000 could support more interfaces, this could be used as an APP output path also.

Chapter 5 Client-server Software for HIPPI Access to Sun File Systems

Introduction

A major goal in IPC selection was to minimize all real-time processing at the Suns, because SunOS Version 4.1 performs poorly on real-time applications. Another goal was to simplify IPC. Therefore, early in the development of the Beamformer, file I/O was given high priority for the APP-to-Sun communication. At that point in system development, the amount of time available for I/O during each 218 millisecond period was unknown. In addition, it was believed that final spectral data products would be formatted in the APPs. This would not require any processing on the Suns for data products with a final destination to disk. Data products to be transferred using the DRP would simply require a Sun process to read and send the data from file. This would allow the disk to act as a large FIFO device for the DRP data.

Characteristics of file IPC

Using a common disk file for IPC has several desirable characteristics. Disk access is one of the most common program operations. Therefore, this form of IPC is very easy to implement for most programmers. In addition, most high-level operating systems provide a protection mechanism that allows multiple processes to share files. These access mechanisms usually allow group access so that access to a given file or directory may be granted to one group of processes while access is restricted for other processes. This type of IPC is persistent because the files exist over process creation, termination and system

restart. Also, the files used for IPC may be "snooped" using system tools to determine what information is being transferred. For example, on a UNIX system "od" or "more" commands can be used to look at the contents of the file. This method of IPC can also serve across multiple processors. This is possible because most operating systems allow near transparent access to common file systems by multiple processors. For example, on most UNIX systems the NFS facility is used to share file systems.

Some disadvantages to disk I/O as an IPC method include speed limitations and write access control. Since disk I/O involves transfer of data to secondary storage, it is much slower than a method that transfers data through primary memory. Also, throughput may vary dependent upon the disk caching mechanisms used. When multiple processors or threads are writing to the same file, coordination can be a problem since two or more processes may write to the same location leaving the file data corrupted. Usually a lock mechanism must be implemented that allows an atomic message to be written by one process before another process is allowed to access the file.

An example of one common use of file I/O for IPC is the Berkeley UNIX printing system [Stevens, 1990]. This print spooler system transfers the data to be printed through files and uses a file for assignment of sequence numbers. A file of the form /usr/spool/lpd/.seq is used by multiple line printer daemon (lpd) processes to get a sequence number for a print job. The lpd process opens a file to get a sequence number (a file of the form

“.seq”) and uses the function *flock()* to get an exclusive lock on the file. It then gets a sequence number, increments the number, and closes the file.

Software development

Due to the limitations of the APP file access services, the author designed software to access the Sun file systems from the APPs using the HIPPI links. As stated in Chapter 1, the Cray APP has three I/O channels: one host disk I/O using the VME link and two HIPPI interfaces. The APP systems allow one asynchronous I/O request on disk access using the VME interface. Many asynchronous requests are allowed on each HIPPI interface. Since a minimum of five asynchronous requests would be required for each cycle to provide all the required products, a method for performing file I/O over the HIPPI links was required. While most super computers allow access to disk systems over HIPPI links, the APP operating system does not include this capability. Also, no other operating system options were available for the APPs. The developed client-server software consists of a library of APP functions that communicate with server software running on a Sun computer. The server software performs the file I/O.

First attempts to develop simple client-server code revealed two characteristics of the HIPPI links that made the design more complex than had been expected. First, it was found that the Sun HIPPI reads would not terminate until the number of bytes specified in the read was satisfied. This occurred even if the HIPPI transfer completed. With most standard I/O devices, a read will complete when end of file is read and the actual number

of bytes transferred is returned by the *read()* function. This HIPPI characteristic required the development of a protocol where the size of the each HIPPI transfer was known apriori for different size file reads and writes.

The other characteristic that complicated the design was how the Sun HIPPI driver processed UNIX *readv()* and *writv()* functions[Stevens, 1992]. These functions allow the user to specify a number of vectors of lengths and addresses. The first version of the server was designed to use these functions to separate headers from data. The Sun HIPPI driver did not allow this.

A working client-server software package was developed by the author that provides the client functions listed and discussed in Table 7. The functions that are not to be used during the time the system processes real-time data, such as *HipOpenHostFile()*, were implemented as synchronous operations. Data transfer functions were implemented asynchronous. *HipHostFileCmd()*, *HipHostFileWrt()*, *HipHostFileRd()*, *HipHostSkWrt()*, and *HipHostSkRd()* functions perform two asynchronous transfers to HIPPI devices, one *awrite()* to send the command to the server and one *aread()* to receive the response.

Two methods are available to perform reads and writes against server files after they are opened. To write using the first method, one sends a write command with *HipHostFileCmd()* followed by a *HipHostFileWrt()*. In this case the first call tells the server that the next transfer will be data to be written to file and how large the packet will

Table 7 APP HIPPI file access client functions

Function	Description
<i>HipHostOpen()</i>	Open HIPPI channel to a Sun host for disk I/O. This function blocks until completion.
<i>HipHostClose()</i>	Close HIPPI channel to Sun that was opened with <i>HipHostOpen()</i> . This function blocks until completion.
<i>HipOpenHostFile()</i>	Open a file on the server. Mode for opening is specified. This function blocks until completion.
<i>HipCloseHostFile()</i>	Close a host file opened with <i>HipOpenHostFile()</i> . This function blocks until completion.
<i>HipHostFileCmd()</i>	Sends a file command to the host. Commands include SEEK_SET, SEEK_CUR, SEEK_END, READ, and WRITE. The READ and WRITE commands specify parameters to be used by a <i>HipHostFileRd()</i> and <i>HipHostFileWrt()</i> function call which must immediately follow. This is an asynchronous function.
<i>HipHostFileWrt()</i>	Sends data for file writes, must follow a <i>HipHostFileCmd()</i> function. This is an asynchronous function.
<i>HipHostFileRd()</i>	Reads data for a host file read request. Must follow a <i>HipHostFileCmd()</i> function.
<i>HipHostSkWrt()</i>	Sends a command to the server to perform a seek and write and sends the write data.
<i>HipHostSkRd()</i>	Sends a command to the server to perform a seek and a read and reads the data from the server.
<i>HipHostRbWait()</i>	Used to wait on completion of asynchronous functions.

be. The client receives a status back from the server for both the command and the data transfer. This transfer is illustrated in Figure 11. Reads using the first methods are similar; a *HipHostFileCmd()* is followed by a *HipHostFileRd()*. The read protocol is shown in Figure 12.

The second read/write method is to use the *HipHostSkRd()* and *HipHostSkWrt()* functions. These functions send the commands to the server and also send or receive the file data. The *HipHostSkWrt()* function sends the information including the host file descriptor, seek position and size in one transfer and follows this transfer with the data packet. The *HipHostSkRd()* functions sends a command to the server that includes file descriptor, seek position and size and then starts an *aread()* to receive the results. Transfers using the *HipHostSkWrt()* and *HipHostSkRd()* functions are illustrated in Figure 13 and Figure 14, respectively.

The functions that combine the command, seek parameter and data in one operation provide the most efficient transfer method. However, using the *HipHostFileCmd()* function provides an acknowledgment of the command separate from the data transfer and was useful for development. The seek and transfer function cannot provide very much information to the user on failures, because only one status is provided back to the user for both the seek and read/write server locations. This is acceptable for the production system, because, if a failure occurs, the real-time data flow will be disrupted requiring that the data processing be restarted.

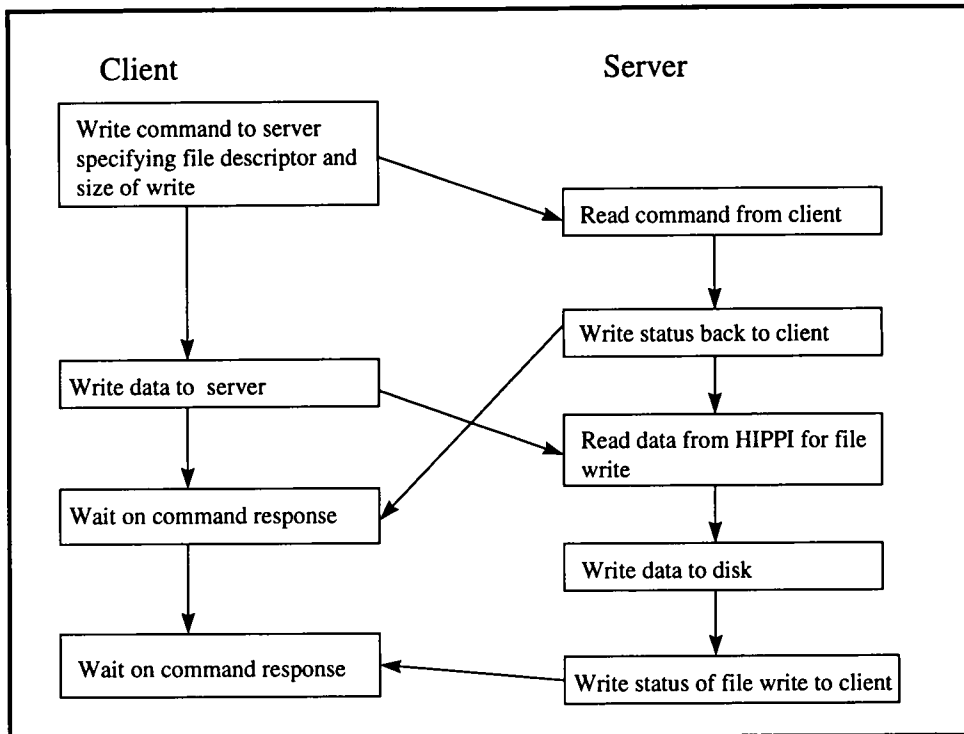


Figure 11 Write using two commands

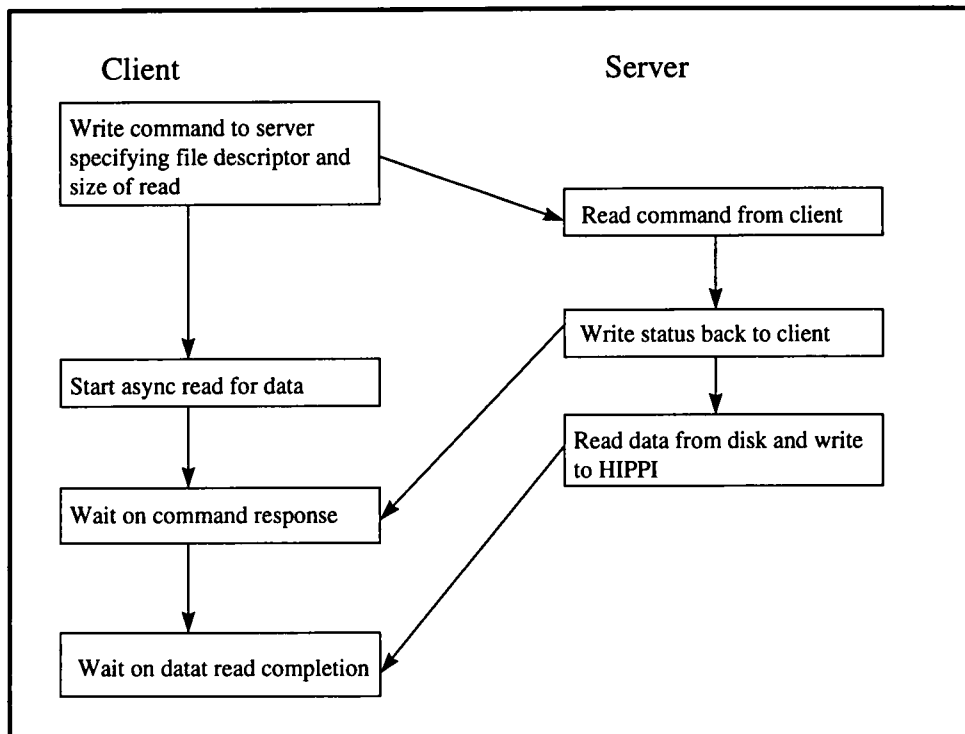


Figure 12 Read using two commands

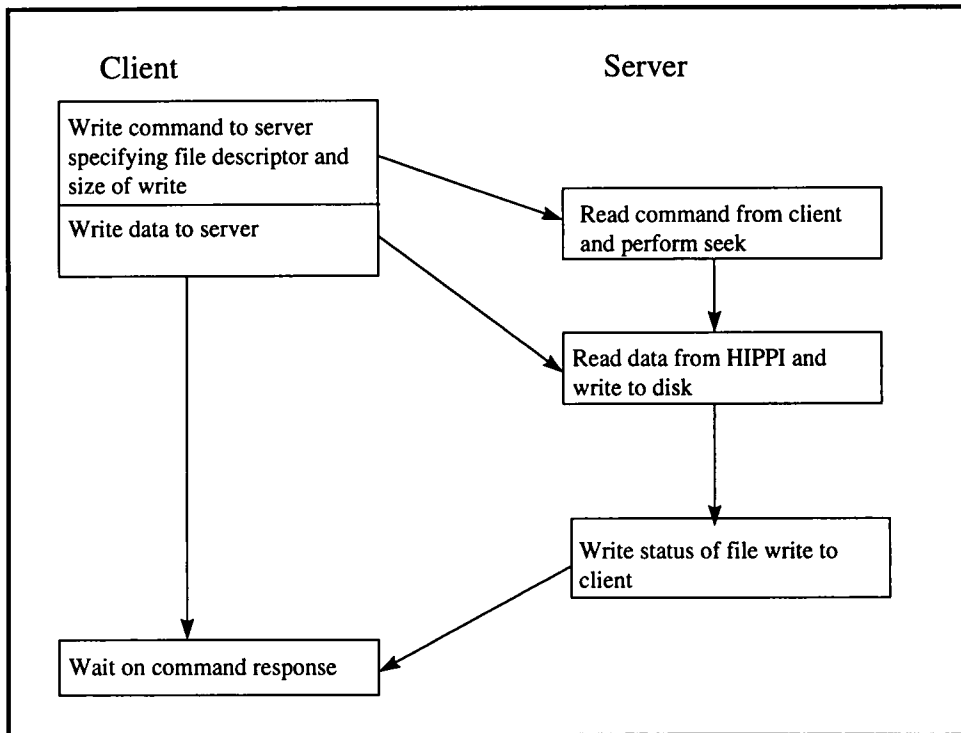


Figure 13 Writes using *HipHostSkWrt()*

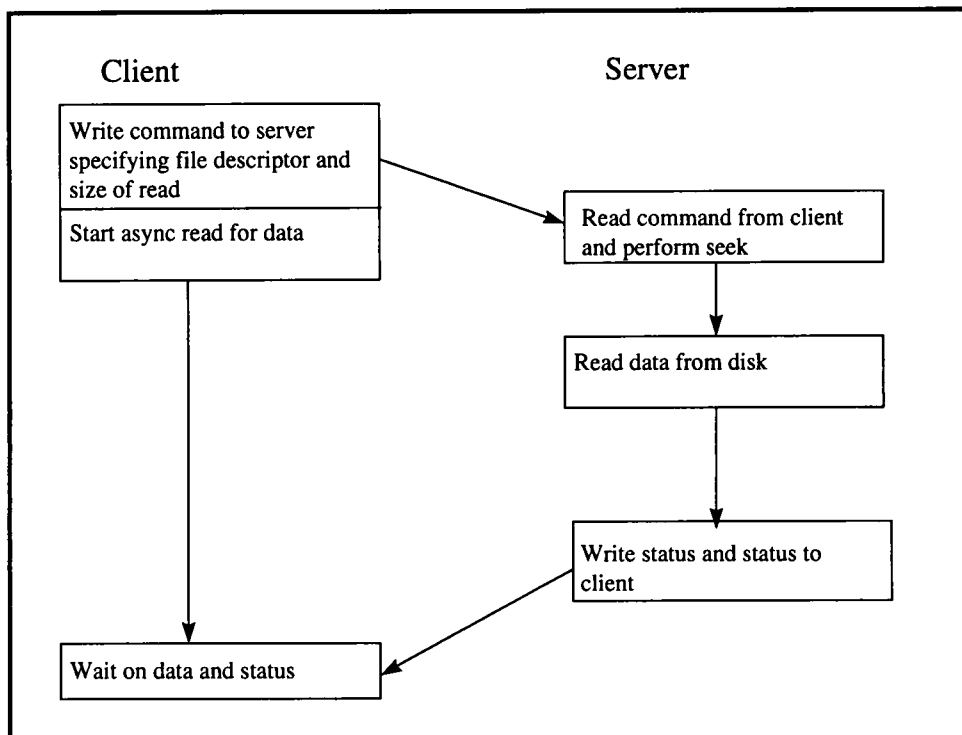


Figure 14 Reads using *HipHostSkRd()*

This client-server pair functioned properly when tested. However, by the time the client-server software was completed, the APP beamforming code was progressing, and the APP characteristics were better known. Experience with the APPs led to system design decisions that eliminated the need for the file access from APP software. It was found that scalar code ran on the APPs at about the same rate as on the Suns. A large portion of the 218 ms frame period would be required to perform the output data product formatting and storage on the APPs. The design decision was then made to send the data products to the Suns in packets containing all the data products and a single copy of the header information. The Sun could be used to separate, store, and send the data to the final destinations. As a result, the APPs were freed to perform other vector-intensive tasks.

Conclusions

The client-server disk access software was never optimized and put into production use. However, its development provided valuable research results. The author discovered the unique characteristics of the HIPPI interfaces and how to develop good HIPPI interface code. It was found that, if transfers did not complete properly, often the reading process on the Suns would become hung in the D-state. The D-state is an uninterruptible process state. A process hung in this state requires rebooting the node. Another valuable lesson learned was that *readv()* and *writv()* operations did not work as expected. Also, experience was gained in using the raw and framing protocol modes for transfers. If development of this code had continued, performance tests would have been performed, and a version with a stateless server may have been designed to compare performance.

Though the client server code was not used in production, it provided a vehicle by which the author learned to use the HIPPI links and paved the way for future code development.

Chapter 6 Design for Beamformer Data Communications

Introduction

After the data communications facilities were characterized, the architecture for data transfer between the beamformer processes was designed. This included a portion of the specifications for the Sun processes that receive spectral products from the APPs. This chapter presents the initial design for the data communications architecture, problems discovered during testing and the final design after corrections were made.

Initial design

One goal of the software design was to develop code portable between nodes. The reason for this goal was to allow load balancing. For example, if a Sun node was overloaded, one of the processes could be moved to a less stressed node without code modification. To allow Sun processes to be moved between nodes with minimal changes, DRP was chosen as the preferred IPC for beamformer processes. This was because DRP does not use node information. However, it was realized that DRP would not be suitable for all IPC. Shared memory was specified for IPC that must take place at the frame rate (218 ms). The author permitted the use of other IPC methods at the discretion of the engineers developing the modules.

Three data paths are available for communications between the APPs and Suns. They are the TMS shared memory, Sun disk files and the HIPPI links. Based on the code delivered

by CRAY, a flag area in TMS SAM-2000 memory was chosen to pass command and status information between the Sun 670MP and the APPs. This flag area was also selected for data synchronization between the APPs and processors in the TMS SAM-2000 system that pre-process the input data. Due to the data rate requirements, the HIPPI links were used to transfer coefficients from APP0 to APP1 and APP2 and send output data from the APPs to Craya1 and Craya2. The HIPPI links were also chosen to transfer the steering data from a Sun node to APP0. An alternative to this would be to send the steering data from Craya to APPs via the TMS SAM-2000 memory, but this would not allow the process to be moved to another Sun node if that became necessary. Data rates required that one Sun SS10 be used for processing, output and storage of digital spectral data and the other SS10 be used to drive the DAC. Craya was chosen as the control node because it manages APP program execution and has access to SAM-2000 memory. Craya was also chosen as the node to run the tracking process, because it was configured with the necessary serial ports. Running the tracking process on one of the Sun SS10s would require moving the serial port hardware from the 670MP to the SS10. Moving the serial port hardware is not a big task, but the author wished to avoid that if possible.

HIPPI channels from all APPs and Suns are connected to a Network Systems HIPPI switch. This allows any of the nodes to send or receive data from any other node forming a high-speed LAN. To allow the processes to create multiple logical connections for the different data transfers, unique framing protocol numbers were used for each transfer

type. These numbers were chosen from the user defined range specified in the ANSI HIPPI standards.

Figure 15 shows a data flow diagram for the initial design. Table 8 gives a description of the processing functions in the figure; Table 9 describes the logical channels that use the HIPPI links; and Table 10 describes the DRP logical links (message types). The only other communications paths are the low speed serial links used to receive steering data, the direct link from Craya2 to the DAC subsystem, the HIPPI links from the APPS to the SAM-2000 and the link from Craya to the TMS SAM-2000 system. The term "processing function" is used here because the APPs only execute one task at a time. The three different processing functions shown in APP1 are part of a single task.

The APP-to-SAM-2000 HIPPI links are dedicated to passing sensor data from the SAM-2000 to the APPs. CRAY supplied a driver to enable APP code to access the SAM-2000 memory as a random access device. Therefore, this data path was not part of the communications design and will not be discussed further.

Since a busy switch output port can cause requesting inputs to wait, the total time available for a HIPPI transfer may be exceeded. For this design, where Craya1 receives packets from all APPs, an APP wishing to send to that port may block keeping its output port busy. The critical situation for this design is when APP0 must wait on the port to Craya1. This occurs when APP1 and APP2 are sending products to Craya1 and APP0 is

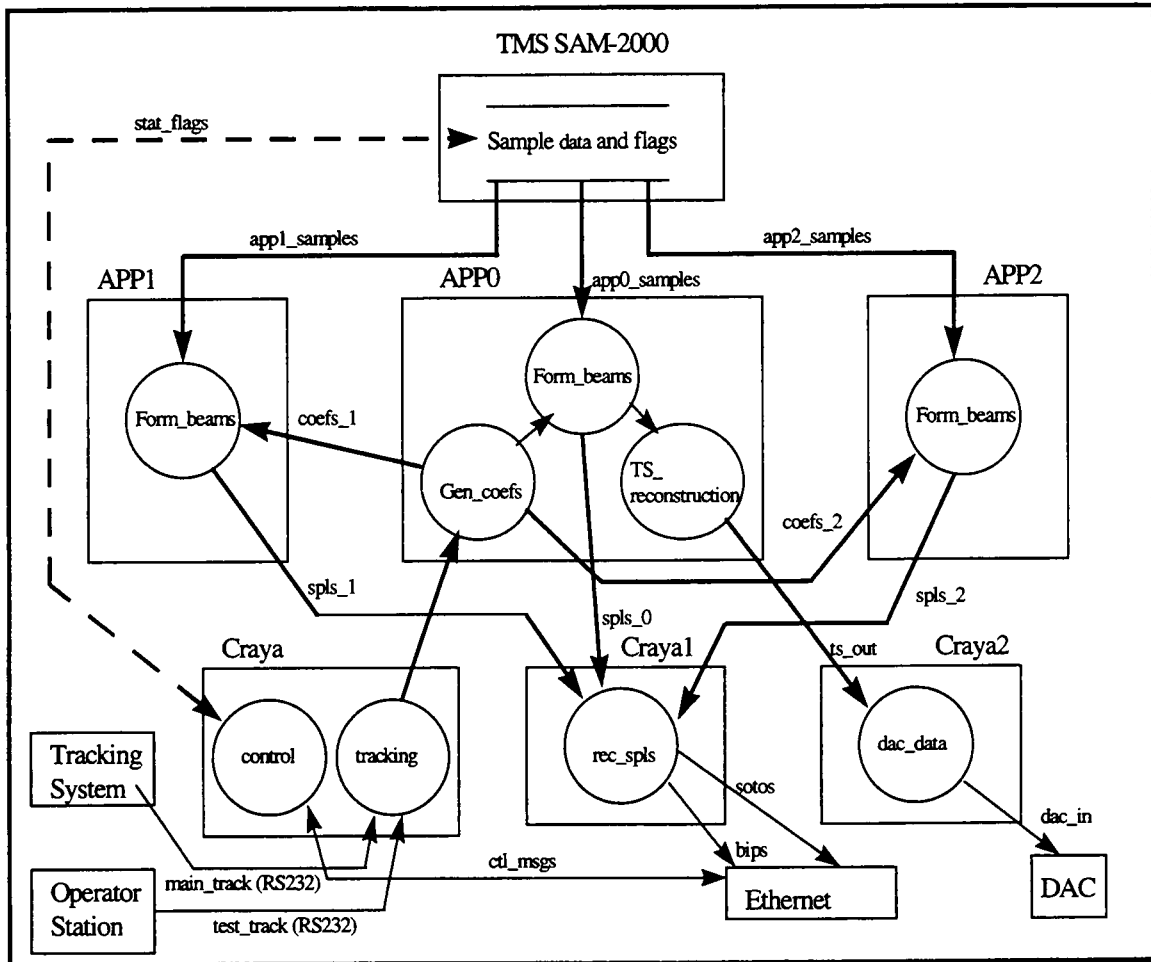


Figure 15 Initial design data flow diagram

Table 8 Processing functions for initial design

Processing function	Description
Form_beams	This includes beam formation and calculation of all spectral products.
Gen_coefs	Generate coefficients. This function executes on APP0 and generates coefficients for all apertures
TS_reconstruction	Time series reconstruction. This code executes on APP0 and reconstructs time series data from the frequency domain samples for DAC output.
Control	This is the system control process. This process starts, stops and controls APP programs and processes running on Craya1 and Craya2 in response to commands from the operator station. Status and response messages are also sent to the operator station.
Tracking	This process receives beam steering information from the operator station and tracking system, pre-processes the data and sends it to APP0 coefficient generation code. For normal tracking, the tracking system input data stream is selected. For calibration, the test_track data stream from the operator station is selected.
Rec_spls	Receive sound pressure levels. This process receives data packets for each aperture. For example, APP0 sends packets for apertures 1 and 2 , APP1 sends packets for apertures 3 and 4 and APP2 sends packets for aperture 5. Each packet contains BIP, SCNB and SOTO data. Rec_spls processes the data, combines data products from all apertures and outputs the results. BIPs and SOTOs are sent to external systems using Ethernet and DRP, and SCNB data is written to disk.
Dac_data	Receives time series samples from APP0 and sends them to the DAC subsystem.

Table 9 HIPPI data channels

Transfer type	Packet size (Bytes)	ULP number	Packet transfer time (ms)
coefs_1	1,925,152	132	26.2
coefs_2	1,064,992	133	14.9
spls_0	184,960	131	34.2
spls_1	184,960	131	34.2
spls_2	92,400	131	17.1
ts_out	45,880	137	41.4
track_out	2048	139	1.3
TS1	196,624	134	3.3
TS2	262,160	135	4.0

Table 10 DRP messages

Transfer type	DRP message types	Packet size	Average data rate
ctl_msgs	bfx_cmd bfx_resp	3	Less than 10 Bps
bips	bfx_bip_data	49,056	225,000 Bps
sotos	bfx_oto_sti_data	1,536	7000 Bps

wanting on the port. This extends the APP0 HIPPI output time, worst case, to the total of the time required to send all its packets plus the time for APP1 and APP2 to send their packets to Craya1. Therefore, the next step in the design was to calculate the times required for all HIPPI transmits and receives on each port used. Table 11 gives the total I/O time for all HIPPI ports. The times in Table 11 were calculated by interpolating the data rate for the particular size transfer from the measured data presented in Chapter 3, Table 5, and dividing the transfer size by the data rate. The total HIPPI transmit time for APP0 is,

$$T_{\text{total-app0_out}} = T_{\text{coefs_1}} + T_{\text{coefs_2}} + T_{\text{spls_0}} + T_{\text{ts_out}}$$

inserting the values gives,

$$T_{\text{total-app0_out}} = 26.2\text{ms} + 14.9\text{ms} + 34.2\text{ms} + 41.1\text{ ms}$$

which gives,

$$T_{\text{total-app0_out}} = 116.4\text{ ms.}$$

Adding in the time required for APP1 and APP2 to transfer data to Craya1 gives,

$$T = 116.4\text{ ms} + 34.2\text{ ms} + 17.2\text{ ms}$$

or

$$T = 167.8\text{ ms.}$$

This is less than the 218 ms frame time. Therefore, APP0 should be able to complete all HIPPI writes for this design.

The only node with significant Ethernet loading is Craya1. This node must send SOTOs and BIPs out over Ethernet using the DRP. The total output data rate for Craya1 is the

Table 11 HIPPI port busy times

HIPPI Port	I/O time per cycle (ms)
APP0 out	116.4
APP0 in	8.6
APP1 out	38.2
APP1 in	26.2
APP2 out	21.1
APP2 in	14.9
Sun CrayX1 out	1.3
Sun CrayX1 in	85.5
Sun CrayX2 out	41.4

sum of the BIP and SOTOs rates. BIPs consists of 49,056 byte packets and SOTOs are sent as 1,536 byte packets. The packets are sent at 218 ms intervals on average.

Therefore the Ethernet data rate is

$$\text{Rate} = (1/218 \text{ ms}) * (49,056 \text{ bytes} + 1,536 \text{ bytes})$$

giving a rate of 232 KBps. This is well within the DRP rate over Ethernet.

As the design of the tracking program progressed, changes to the original data communications design were required. It was found that the tracking process must receive data that specifies what frame number is being processed. Options for how to get the real-time frame number included reading the value from SAM memory, receiving the value over the HIPPI link or getting it from the rec_spls process. The SAM memory option was rejected, because another task on Craya was accessing the memory area for status information and asynchronous reads of the SAM memory had not been tested. The second option was rejected, because the author did not want to add another HIPPI

transfer. The option of getting the value from the rec_spls process which receives the frame number with the data products was selected. Since the single task processing on the APPs is a deterministic pipelined process, the number for any frame in the pipeline can be determined from the output frame number. The design of the rec_spls and tracking processes were modified to transfer the frame number from rec_spls to the tracking process through shared memory. This required moving the tracking process from Craya to Craya1. The design change included the use of a UNIX signal to trigger the tracking process to read the new frame number written by the rec_spls process. The rec_spls process reads the Process-ID (PID) of the tracking process from shared memory to know which PID to send the signal to. This PID is written by the tracking process at start-up.

With the exception of moving the tracking process to Craya1, the system was first implemented with the communication design presented earlier in Figure 15. The system worked as designed, but data transfer problems caused the system to fault intermittently. The faults were traced to a condition where an APP was unable to complete a HIPPI transfer in a 218 ms frame interval. The system ran without the failures when the SPL transfers from the APPs to the Sun rec_spls process were disabled. Investigation showed that the rec_spls process would not always complete the five HIPPI reads in the required time. The problems with the first rec_spls process design are covered in chapter eight in detail, but the root of the problem can be summarized as being related to the lack of real-time support by the SunOS operating system. In an effort to eliminate the faults, FIFOs

were added to both the APP code that sends data to the Sun and to the rec_spls process. These changes improved the robustness of the system, but further improvements were needed.

To provide additional robustness, the design of the APP-to-rec_spls transfers was modified. In the original design, all APPs send each aperture's products to the Craya1 node. This required that all APP output packets to rec_spls be the same size. This packet size is set by aperture one which is larger than needed for the other five apertures. Since aperture one has more data points, space was wasted on the transfers for apertures two through five. The design was modified so that APP1 and APP2 send spectral products to APP0 which sends data for all apertures to the rec_spls process in one packet. This improved the ability of Craya1 to receive the HIPPI packets, because the larger transfers are more efficient and only one read instead of five is required each cycle. The modified data flow diagram is shown in Figure 16. Table 12 describes the new design's HIPPI logical channels. The HIPPI data transfer times for all nodes in the new design are shown in Table 13.

Measurements

Measurements of the actual time required for the APP0 HIPPI output operations were taken. Measurements were made by inserting a breakout box in the HIPPI cable paths and recording the state of the HIPPI "connect" signal with a Hewlett Packard logic analyzer. Figure 17 shows the timing for the APP0 output along with the SPL and time

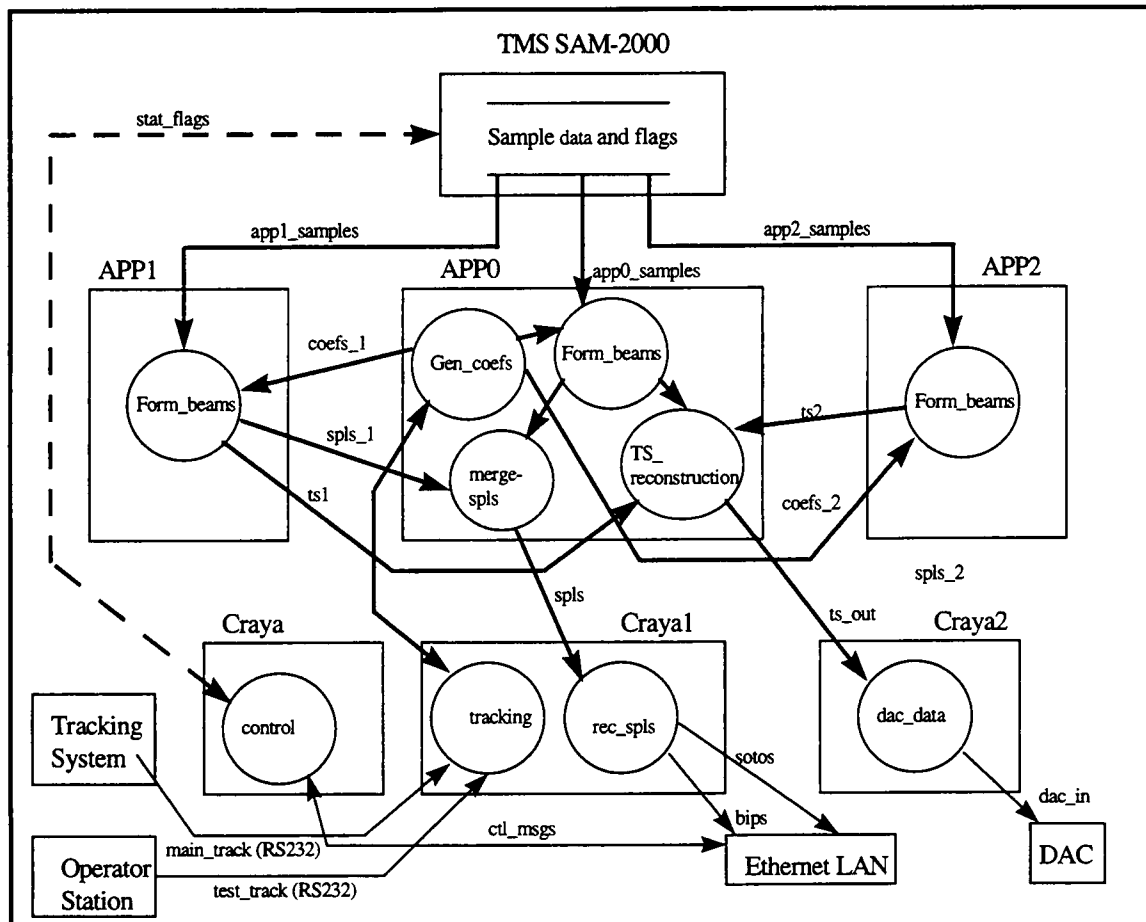


Figure 16 Final design data flow diagram

Table 12 New design HIPPI channels

Transfer type	Packet size (Bytes)	ULP number	Packet transfer time (ms)
coefs_1	1,925,152	132	26.2
coefs_2	1,064,992	133	14.9
spls_0	233,136	131	27.1
spls_1	79,008	131	3.2
spls_2	39,504	131	2.4
ts_out	45,880	137	41.4
track_out	2048	139	1.3
TS1	196,624	134	3.3
TS2	262,160	135	4.0

Table 13 HIPPI port busy times for revised design

HIPPI Port	I/O time per cycle (ms)
APP0 out	109.6
APP0 in	14.2
APP1 out	6.5
APP1 in	26.2
APP2 out	6.4
APP2 in	14.9
Sun Craya1 out	1.3
Sun Craya1 in	27.1
Sun Craya2 in	41.4

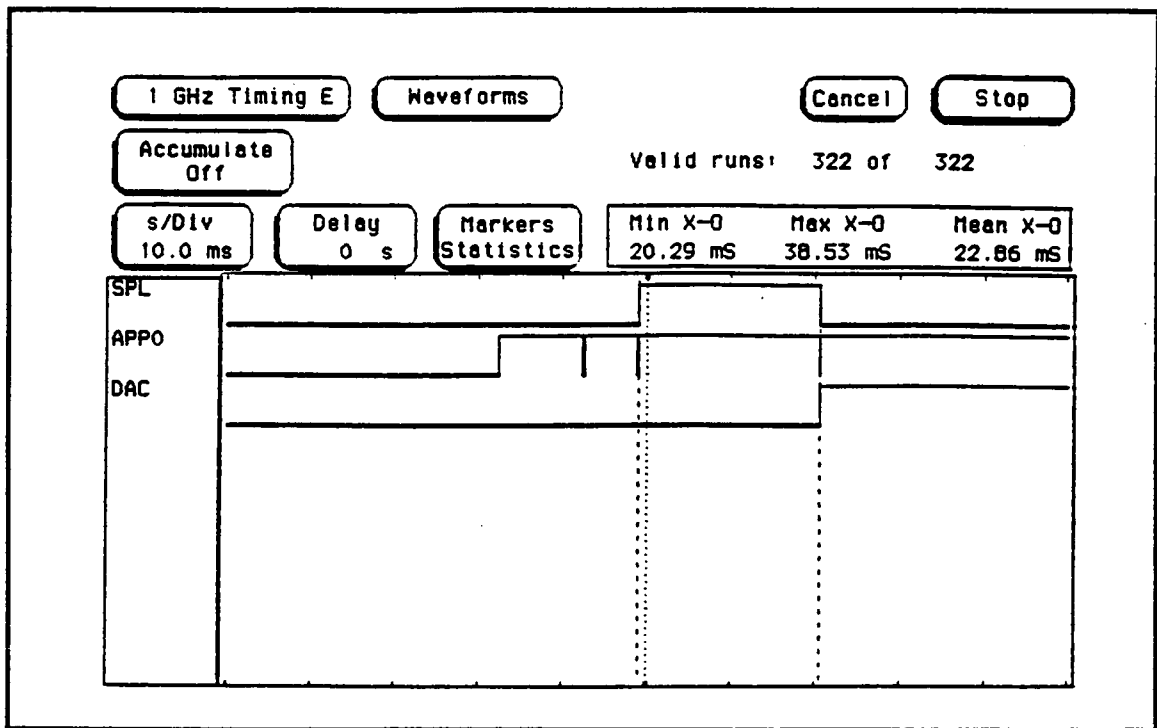


Figure 17 Original APP0 HIPPI output timing

series (DAC) components. The Figure 17 measurements were made while running the version of the software that performed SPL transfers from each APP to Craya1. Figure 18 shows the SPL and time series (DAC) timing for the final version of the software where all SPLs are sent to the Sun computer from APP0. The measured values agree well with the design calculations.

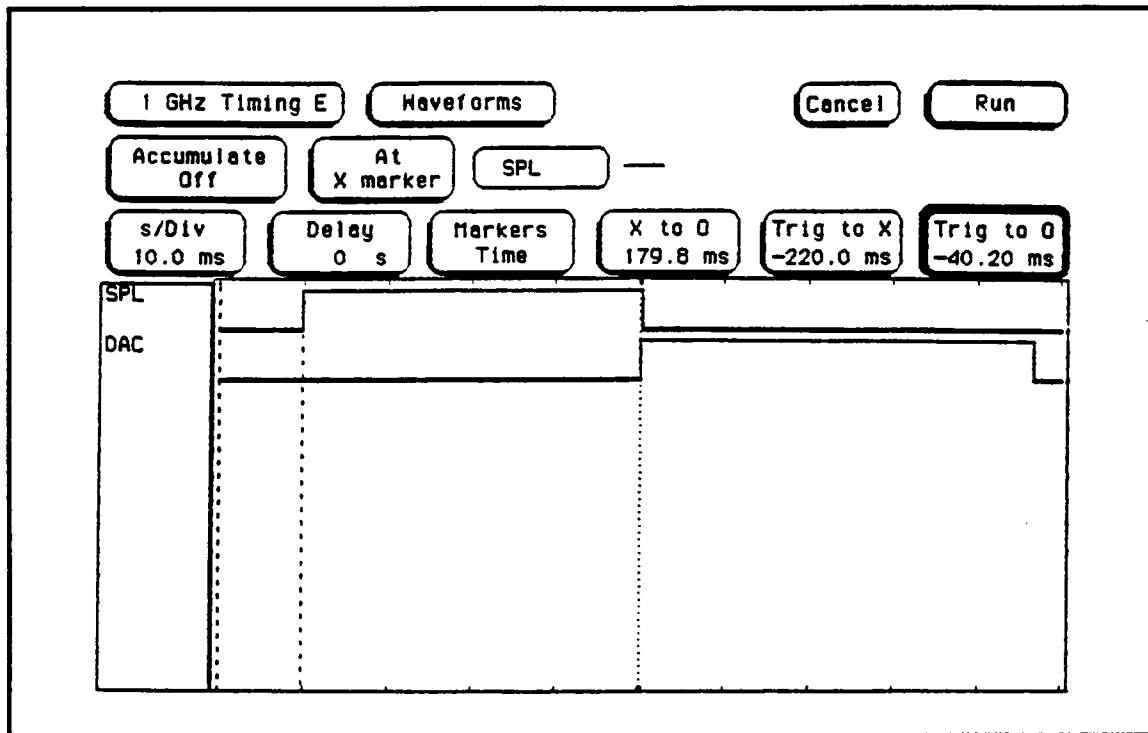


Figure 18 Final design APP0 HIPPI output timing

Chapter 7 Library for APP HIPPI Communications

Introduction

This chapter describes a library of APP FORTRAN-callable functions (Ahipio) to perform HIPPI I/O. The library functions isolate details of the HIPPI I/O from the signal processing code. While it is possible to incorporate this I/O directly into the signal processing code, the use of this library provides the following advantages

- CRAY recommends that I/O be done from C code for best performance. FORTRAN wrappers are provided, but by developing the library, application specific functions could be added.
- Specific functions for each data transfer type are provided.
- The increased modularity allowed parallel code development.
- The signal processing code is simplified.

Code description

Functions included in the Ahipio library are listed in Table 14. Most of the functions are one of four types: “open..”, “close..”, “send..”, or “rec..”. The “close..” functions are simply wrappers for a C *close()* function. Likewise the opens used to perform reads simply open a file descriptor and check for errors. The opens used for write operations are more complex. The device is opened, and an *ioctl()* function is used to set I-Field parameters. The I-Field is a 32-bit word that specifies a ULP and the “camp on “ bit.

Table 14 Ahipio functions

Function	Description
<i>open_ts_10_wr()</i>	Open file descriptor for time series writes to APP0 on APP0.
<i>open_ts_20_wr()</i>	Open file descriptor for time series writes to APP0 on APP2.
<i>open_ts_10_rd()</i>	Open file descriptor for times series reads from APP1 on APP0.
<i>open_ts_20_rd()</i>	Open file descriptor for times series reads from APP2 on APP0.
<i>open_oa_21_wr()</i>	Open file descriptor for overlapped aperture data writes to APP1 on APP2.
<i>open_oa_10_wr()</i>	Open file descriptor for overlapped aperture data writes to APP0 on APP1.
<i>open_oa_21_rd()</i>	Open file descriptor for overlapped aperture reads from APP2 on APP1.
<i>open_oa_10_rd()</i>	Open file descriptor for overlapped aperture reads from APP1 on APP0.
<i>close_spl()</i>	Close file descriptor for SPL writes on APP0.
<i>close_steer()</i>	Close file descriptor for steering data reads on APP0.
<i>close_ts_sun()</i>	Close file descriptor for time series writes on APP0.
<i>close_ts_00()</i>	Close file descriptor for time series writes to APP0 on APP0.
<i>close_c_01()</i>	Close file descriptor for coefficient writes to APP1 on APP0.
<i>close_c_02()</i>	Close file descriptor for coefficient writes to APP2 on APP0.
<i>close_ts_10()</i>	Close file descriptor for time series writes to APP0 on APP1.
<i>close_ts_20()</i>	Close file descriptor for time series writes to APP0 on APP2.
<i>close_oa_21()</i>	Close file descriptor for overlapped aperture writes to APP1 on APP2.
<i>close_oa_10()</i>	Close file descriptor for overlapped aperture writes to APP0 on APP1.
<i>send_spls()</i>	Send SPLs to Sun node Craya1.
<i>send_status()</i>	Send status packet to Sun node Craya1.
<i>send_ts()</i>	Send time series data to Sun node Craya2.
<i>send_c_01()</i>	Send coefficients to APP1.
<i>send_c_02()</i>	Send coefficients to APP2.
<i>send_ts_10()</i>	Send time series data from APP1 to APP0.
<i>send_ts_20()</i>	Send time series data from APP2 to APP0.
<i>send_oa_21()</i>	Send overlapped aperture data from APP2 to APP1.
<i>send_oa_10()</i>	Send overlapped aperture data from APP1 to APP0.
<i>rec_c_01()</i>	Receive coefficients sent from APP0 on APP1.
<i>rec_c_02()</i>	Receive coefficients sent from APP0 on APP2.
<i>rec_ts_10()</i>	Receive time series data from APP1 on APP0.
<i>rec_ts_20()</i>	Receive time series data from APP2 on APP0.
<i>rec_oa_21()</i>	Receive overlapped data from APP2 on APP1.
<i>rec_oa_10()</i>	Receive overlapped data from APP1 on APP0.
<i>rec_steer()</i>	Receive steering data from Sun node Craya1 on APP0.
<i>await()</i>	Wait on asynchronous operation to complete.

The ULP specifies the destination port on the HIPPI switch. When the “camp on” bit is set to one, a connection request will wait for a switch destination port to become free if it is busy at the time a connection is initiated. If the “camp on” bit is zero (default) and the destination port is busy, an error will be returned by a *write()* function. The source code for a function that opens a HIPPI I/O descriptor for writes is shown in Appendix C.

The “send..” functions are FORTRAN wrappers for *awrite()* functions. The file descriptor is determined by the function and is, therefore, hidden from the user. The APP *awrite()* functions are used for asynchronous writes. The *awrite()* returns a positive request block number when the write is successfully started. The *await()* function must be used to determine when the I/O has actually completed. The request block number is returned to the user to check. An example “send..” function is shown in Appendix D.

The “receive “ functions work the same way as the write functions. They provide a FORTRAN wrapper for the C code, embed the file descriptor and return a request block number.

The *await()* function is the wrapper for the *await()* system function. Two values must be passed to *await()*, a request block number and a mode. If the mode is set to one, the function will block until the I/O for the operation specified by the request block completes. If the mode is set to zero and the I/O is not complete, the function will not block and the status will be set to “-1”. This function is shown in Appendix E.

APP data transmission

The use of asynchronous reads and writes allows data to be transferred in and out of the APP at the same time computations are being performed. The HIPPI I/O takes place under control of a service processor and DMA hardware. Once the I/O operations are started, they run to completion without affecting the signal processing speed. The overlap of processing is illustrated by the flow chart in Figure 19.

During each 218 ms cycle, new I/O operations are started, completion of previous I/O is checked and computations are started on a new set of data. The first version of the APP code used simple double buffering of the output data. During testing, it was found that, at times, the writes to the Suns would not complete in the 218 ms. Therefore, a larger FIFO was implemented to allow for times when the Suns could not accept a new packet. During normal operation, the "N" data products are being sent while the "N+1" products are being computed.

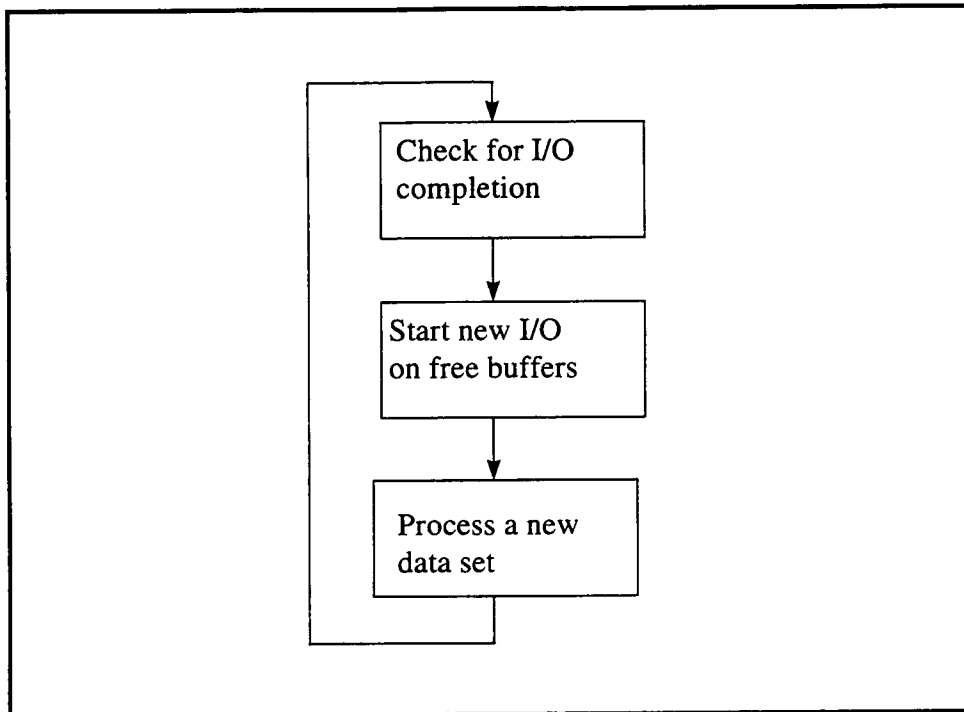


Figure 19 Basic APP async I/O flowchart

Chapter 8 Design of Software to Process APP Products

Introduction

This chapter describes the design of the `rec_spls` program. This program runs on the Craya1 nodes and is responsible for processing the digital output SPL data streams. The original requirements are given first followed by the design history. Details of the version of `rec_spls` delivered with the system are presented in the next chapter.

Requirements

The `rec_spls` process receives data products from the APPs, creates the required output streams and delivers them to the destinations. In the original design, `rec_spls` receives data packets for each aperture. Therefore, each cycle it receives 2 packets from APP0 and APP1 and one packet from APP2. Each packet is the same size. `rec_spls` must combine the spectral products from each operation into three output streams. The three streams are BIPs, SOTOs and CNBs. The BIPs and SOTOs are written to external systems via the DRP, and CNBs are written to disk. The writing of CNBs to disk is under control of a shared memory flag. `rec_spls` must also write the buffer number from the input packets to shared memory for use by the tracking process.

Original Design

A context diagram for the original `rec_spls` design is shown in Figure 20. Table 15 describes the data terminations, and Table 16 describes the data flows.

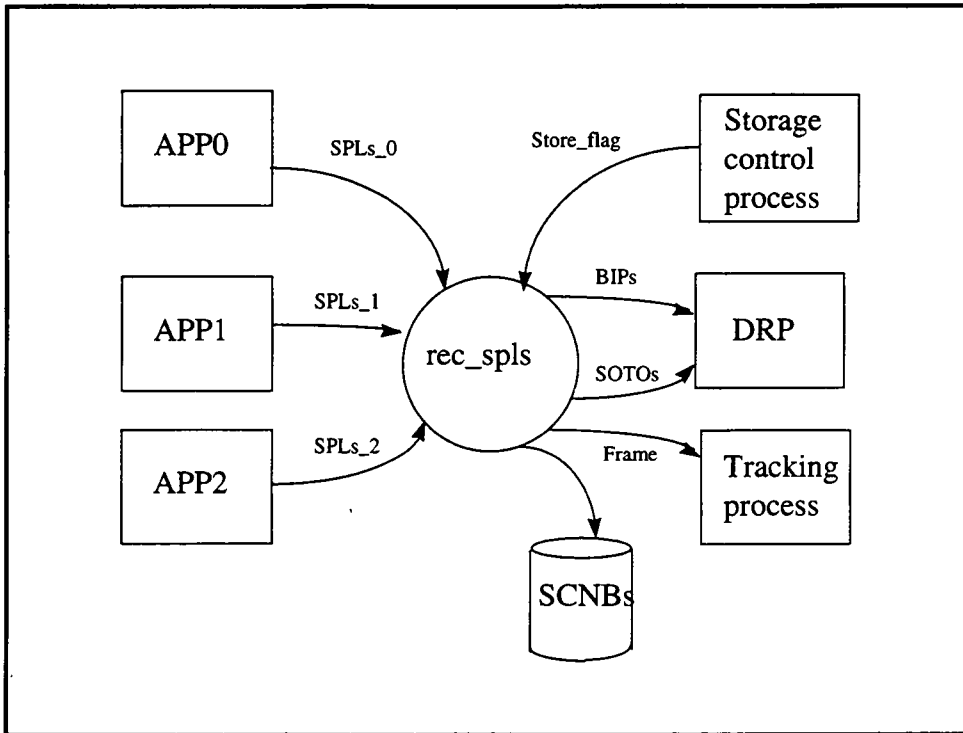


Figure 20 Original rec_spls context diagram

Table 15 Original rec_spls terminator descriptions

Terminator (data sources/sinks)	Description
APP0	CRAY APP that generates the spectral products for apertures one and two.
APP1	CRAY APP that generates the spectral products for apertures three and four.
APP2	CRAY APP that generates the spectral products for aperture five.
DRP	The Data Routing Process. This process sends data via TCP/IP to other processes. Data products from rec_spls are usually read by two other nodes.
Storage control process	This process receives commands via DRP and sets a flag in shared memory to tell rec_spls whether SCNB data is to be stored to disk.
Tracking process	Process that sends steering data to the APPs.

Table 16 Original rec_pls data flow descriptions

Data flow	Description
BIPs	Beam Image Products. This data is sent using the DRP to monitoring and archiving nodes.
Frame_number	An integer in shared memory that contains the frame number from the last packet received from APP0. This is used by the tracking process to generate steering data for the APPs.
SCNB	Short term integrated, Course Narrow Band spectral products. These are written to disk for later processing.
SOTOs	Short term integrated One-Third Octave values.
SPLs_0	Packets from APP0 containing spectral products for apertures one and two.
SPLs_1	Packets from APP0 containing spectral products for apertures three and four.
SPLs_2	Packets from APP0 containing spectral products for aperture five.
Store_flag	Flag to indicate when SCNB data should be written to disk. A new file is started on a 0 to 1 transition.

The original rec_spls was designed as a relatively simple task. After initialization, the process reads the input data, organizes the received data into the output structure and writes the results to the destination. Asynchronous HIPPI reads are used to support buffering of the input data. First a loop is executed that performs five reads to get an entire frame of data. If two data sets from the same aperture arrive before all five are received, the program exits due to the error. After each successful read, the data for the aperture is merged into the three output data structures, two structures for DRP writes and one for a disk write. When all five operations have been received for a given frame, writes to disk and DRP are performed. Only synchronous writes are possible to DRP. Pseudo code for the main code is shown in Figure 21.

The program was tested by writing an APP program to send simulated data packets to the process. One APP was used to send the data for all apertures. This allowed testing with only one APP. The simulation program was written to send data as fast as rec_spls could receive it. This tested the rec_spls average execution time. The test results showed that rec_spls could process a frame of data in about 120 ms. 120 ms is far less than the 218 ms of frame time required for the application. The measurement was made by executing a large number of loops (> 1000) and using the UNIX time function to measure the execution time. The frame processing time was then calculated as:

$$\text{Frame processing time} = \frac{\text{Total Time}}{\text{Number of Loops}}$$

```

main
{
    Initialize variables, open devices and set up signal handler
    Start async read on HIPPI device
    loop until interrupted
    {
        for all active aperatures ( normally one through five)
        {
            wait on HIPPI read completion
            if error on read or data invalid
            {
                execute end_process
            }
            else
            {
                merge data for aperture read into output structures
            }
        }
        write data to disk and DRP
    }
}

```

Figure 21 Original rec_spls pseudo code

While the tests using simulated data indicated that `rec_spls` was working properly, testing with the beamforming code revealed problems. After running for several hundred buffers, the APP programs would fault, indicating that a write to `rec_spls` did not complete in the 218 ms frame time. The number of frames processed before failure varied from under 100 to over 500. It was suspected that, at times, the synchronous DRP writes would take longer than the allowed time. Test programs were written to measure the times required to perform synchronous writes to disk and DRP. The duration of DRP writes were measured in the `rec_spls` task. That data is presented in Figure 22. A simple loop was written to simulate the disk writes and measure disk write block times. The disk block times are shown in Figure 23. Another program was written to measure the time required for synchronous TCP writes. The TCP data is shown in Figure 24. The data verified that, while on average the writes executed in the required time, at times the write execute time was excessive. This example demonstrates how SunOS is not suited for real-time process execution.

For operating system comparison, the disk-write test was also run on a Sun workstation running the Solaris operating system. Solaris allows a process to be run as a real-time task at a fixed priority. On Solaris, the disk test ran slower on average because it was a slow CPU, but the times were very consistent. For the test, the average write time was about 80 ms, but no write exceeded 90 ms during 1000 writes. Due to other system requirements, the author did not have the option of changing to the Solaris operating system. Instead, two major changes were made to `rec_spls` to allow for non-deterministic

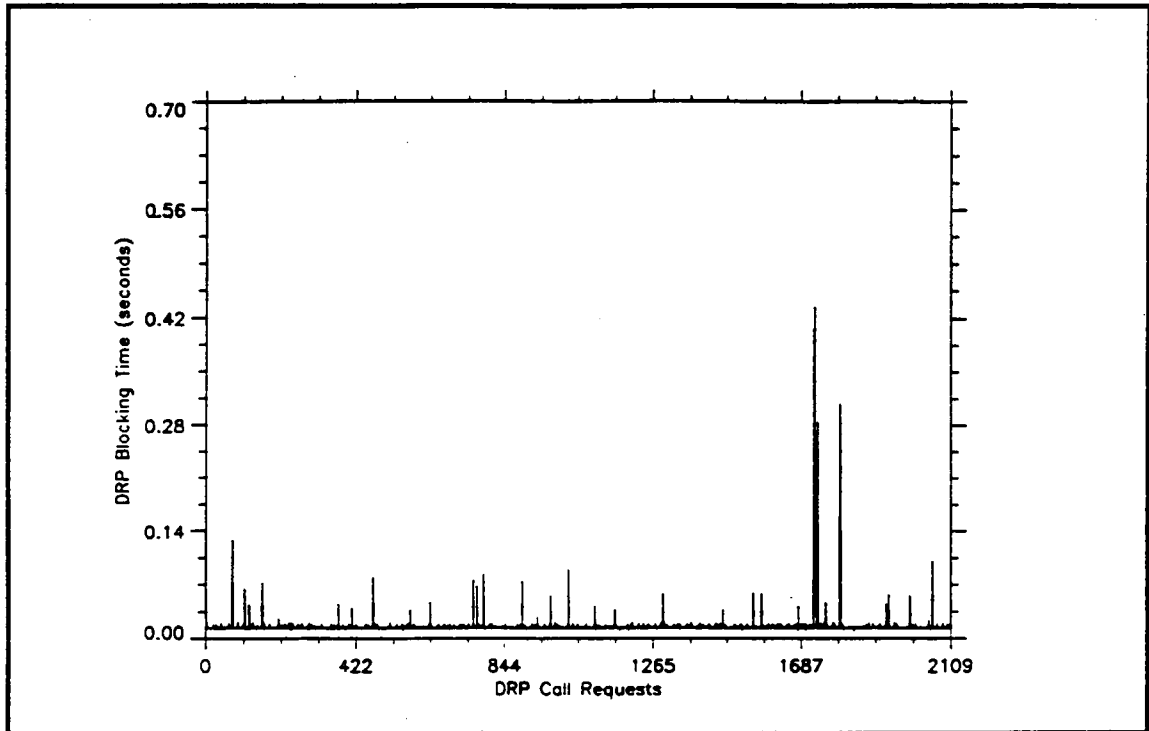


Figure 22 Measured DRP write time

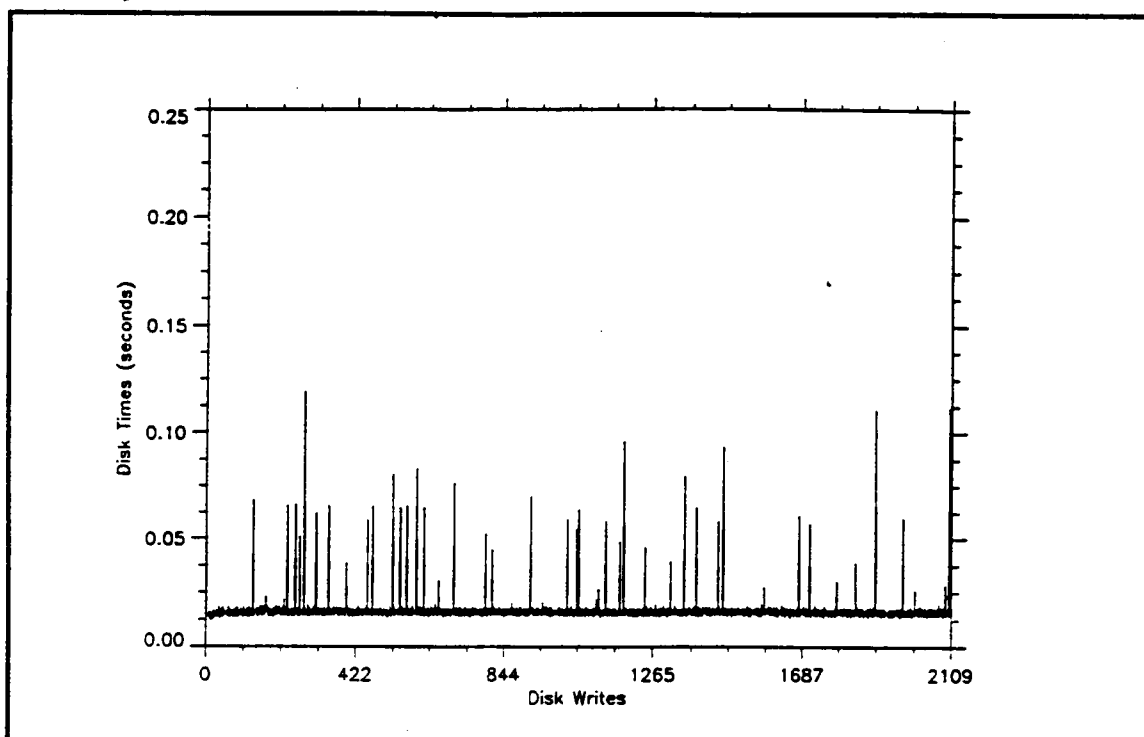


Figure 23 Measured disk write block time

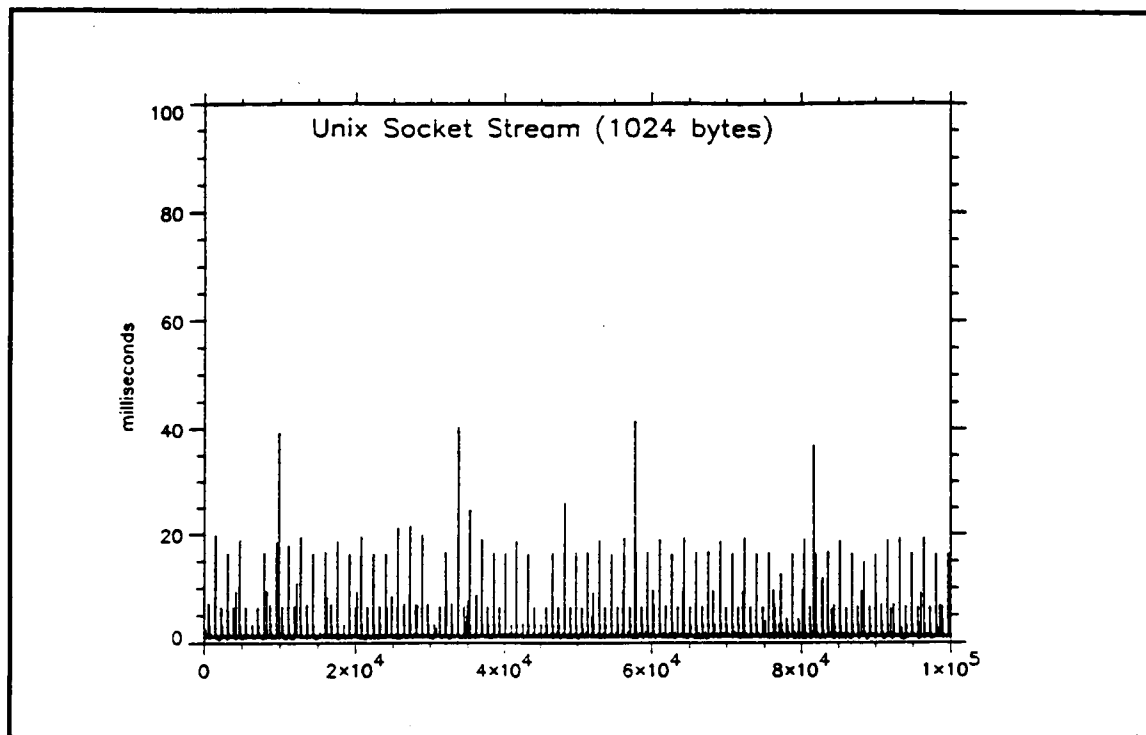


Figure 24 Measured TCP write times

behavior. First, multiple buffers were added to the input data stream (HIPPI reads). Instead of having double buffered reads from HIPPI, an eight deep FIFO was added so that multiple buffers could be received while the program execution was delayed. Second, FIFOs were added to the output data streams. Adding FIFOs to the output streams was complicated by not having the ability to perform asynchronous DRP writes. The FIFOs were implemented using multiple processes and shared memory. The additional buffering allowed the APP code to run without hold-ups on data transmission to rec_spls. Details about FIFO implementation are given in Chapter 9.

Another problem that surfaced during testing was that at random times the rec_spls main process would fail because of a segmentation fault. This problem was traced to a pointer that was being set to an invalid value. It was found that the HIPPI reads were overwriting the input buffers and corrupting other variables. It was determined that the problem was caused by a defect in the HIPPI device driver. Vendor support was not available, so software was modified to mask the problem. Additional space was allocated to the input buffers so overwrites would not corrupt other variables. With this change, the code was placed into production use.

One other major change to rec_spls was made to increase system robustness. Sending data from all three APPs to Craya1 with the same packet size wasted bandwidth, so the data flow was modified. The new data flow design uses APP0 to receive data from APP1 and APP2. APP0 then combines the data into a single packet that is sent to Craya1.

While this involves double handling some of the data, the data on the slow line to Craya1 is minimized. This was the last major change made to rec_spls. The program has been in production for about two years and is working properly.

Chapter 9 Details of the Process to Receive Spectral Products

Introduction

One of the two primary data output streams from the Beamformer uses the `rec_spls.c` program which runs on the Craya1 Sun node. The program name, `rec_spls.c`, is short for "receive sound pressure levels." This program is responsible for receiving data packets from APPs, formatting the data, and sending it to the proper destinations. The data streams processed by `rec_spls` includes beam image products, short term integrations of the one-third octave magnitudes, and coarse narrow band data. These data products are described in Chapter 1.

Interfaces and Specifications

A context diagram for the `rec_spls` process is shown in Figure 25. Table 17 describes the terminations, and Table 18 describes the data flows. Additional specifications include

- runs as an interactive process for debugging;
- runs detached for production use;
- tolerates error conditions that cause output streams to fail including disk full conditions and failure of the DRP to handle the required data rates; and
- allows for out of order asynchronous HIPPI read completion.

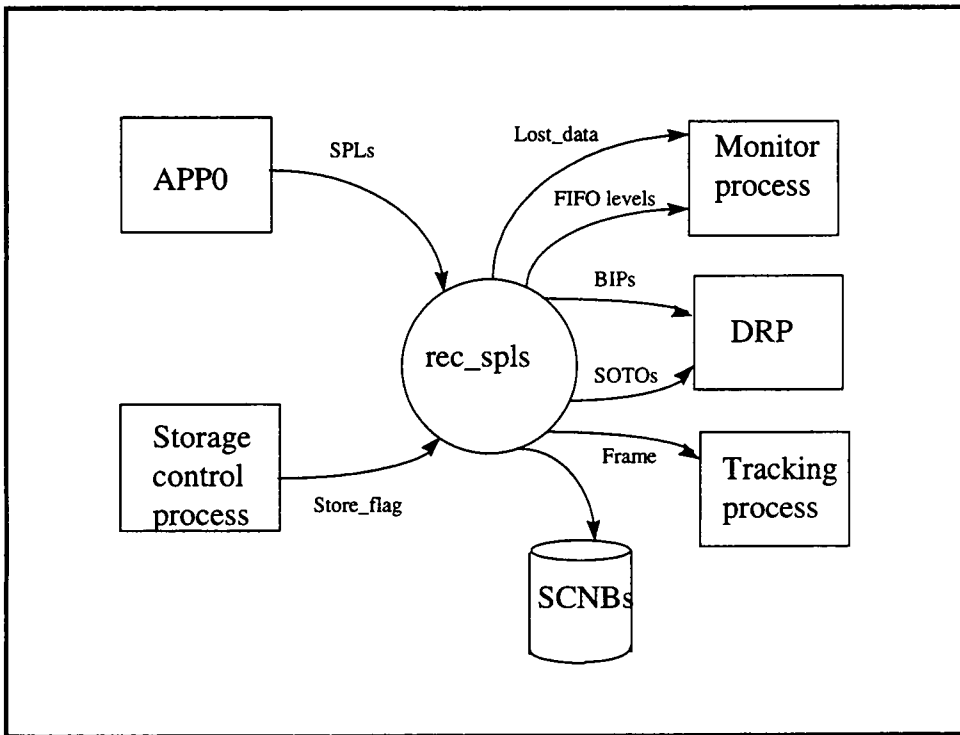


Figure 25 **rec_spls** context diagram

Table 17 rec_spls terminator descriptions

Terminator (data sources/sinks)	Description
APP0	Cray APP that sends the spectral products for all 5 apertures.
DRP	The Data Routing Process. This process will send data via TCP/IP to other processes. Data products from rec_spls are usually read by two other nodes.
Monitor process	Process to display several system variables on the operator console.
Storage control process	This process receives commands via DRP and sets a flag in shared memory to tell rec_spls whether SCNB data is to be stored to disk.
Tracking process	Process that sends steering data to the APPs.

Table 18 Data flow descriptions

Data flow	Description
BIPs	Beam Image Products. This data is sent using the DRP to display and archiving nodes.
FIFO levels	Values in shared memory that are displayed by a monitoring process on the system console. These values specify how full internal buffers are.
Frame	An integer in shared memory that contains the frame number from the last packet received from APP0. This is used by the tracking process to generate steering data for the APPs.
Lost_data	Values indicating the number of data sets that were lost due to a FIFO full condition. FIFO full occurs if data cannot be sent to the DRP or disk at the required rate.
SCNBs	Short term integrated, Course Narrow Band spectral products. These are written to disk for later processing.
SOTOs	Short term integrated One-Third Octave values.
SPLs	Packets from APP0 containing all spectral products
Store_flag	Flag to indicate when SCNB data should be written to disk. A new file is started on a 0 to 1 transition.

Rec_spls normally runs on Craya1 and is started by a remote procedure call from Craya. Command line options are given in Table 19. After startup, a data stream from APP0 is expected to begin within 30 seconds. If the input data does not begin within the specified time, rec_spls exits. After the data stream starts, rec_spls will exit if the input data stops for five seconds. This is the normal exit mechanism. Both the 30-second and 5-second time-outs may be easily changed by modifying definitions in the code.

Program architecture

In order to support Beamformer code running on the APPs, the rec_spls process must be able to receive a data packet every 218 ms. Since the process execution and the data destination streams (disk and DRP) are not deterministic, multiple buffers are necessary. FIFO buffers are implemented for SCNB writes to disk and for SOTO and BIP writes to DRP. The FIFOs may be viewed as a group of boxes. In this view, each box holds one buffer of data to be sent to a destination. Processes are forked to read data from the FIFOs and write it to the destination. The FIFOs use shared memory as shown in Figure 26. The child processes are forked from the parent rec_spls and loop in one function until termination. The main process creates shared memory for the FIFO data blocks and for a structure which contains a count of how many data boxes are occupied.

Both the parent and child processes maintain an index into the FIFOs. When the parent writes a buffer to the FIFO, it increments the write pointer to the next frame's data in the new box. When the parent starts a new buffer, it also increments the FIFO level. The

Table 19 rec_spls command line options

Option	Description
-d	Run detached. Normal operation.
-s	Inhibit SCNB writes to disk.
-o	Inhibit SOTO writes to DRP.
-b	Inhibit BIP writes to DRP.
-a [value]	Active aperture mask. One bit for each aperture with aperture 1 the least significant bit.
-m [value]	Print milestones every "value" buffers.

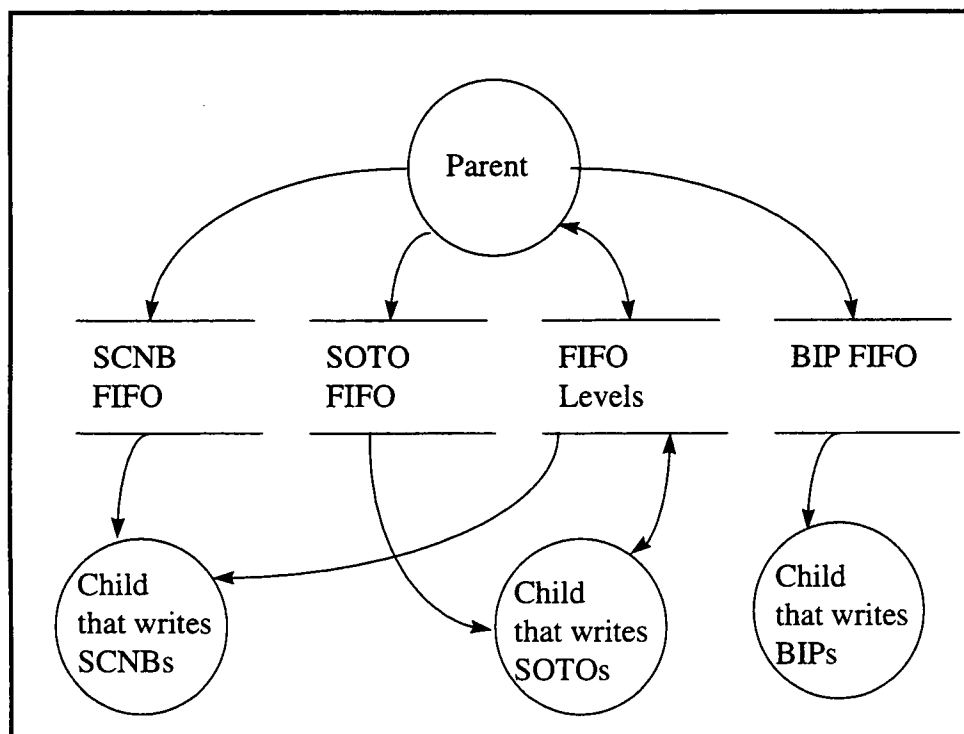


Figure 26 FIFO implementation

level tells the child how many boxes are occupied. If the level is equal to the length of the FIFO, the parent writes the next frame's data to the same box it used for the last frame. This is a data loss situation, and counters are incremented to inform operators of the amount of data lost.

The child processes monitor the FIFO level to see when data is available to send to DRP or write to disk. If the FIFO level is greater than zero, the box pointed to by the FIFO index is written to its destination. After the write, the FIFO level is decremented, and the index is incremented.

The FIFO length is specified by parameters. Separate lengths for the different data products are supported. When it reaches the last box in the FIFO, the index is set back to zero.

The input FIFO is implemented using several buffers with pending asynchronous reads. Each time a read completes a new one is started for that buffer. This allows some elasticity on the receiving end. The APP process that writes the data also uses multiple buffers with asynchronous writes. In this way, if the Sun pauses for a short period (up to approximately 2 seconds), no data will be lost and no processing disrupted.

Detailed program description

The actual source code used to build the `rec_spls` program is described in this section.

The source code files are listed in Table 20. A UNIX “Makefile” is used to build a single executable named “`rec_spls`” from the source files.

The first source files presented are `rec_spls.h`, `spl_products.h`, and `spl_rec_header.h`.

These files are listed in Appendix F, Appendix G, and Appendix H, respectively. The file, `rec_spls.h`, contains common definitions, include files, global variables and several structure definitions. The file `spl_rec_header.h` contains a structure that defines header information for all output products. The file `spl_products.h` contains structures used to build the structure that receives data from the APP.

The `main()` function is in the `rec_spls.c` file which is listed in Appendix I. The first action performed is to get the name of the program from `argv` and initialize logging with `log_init()`. This and all Sun Beamformer programs use the syslog facility for logging. `log_init()` handles the log initialization. The variable `debug` is used to determine whether error and information messages are sent to STDOUT and STDERR. When `debug` is not “1”, messages are only sent to the log file. Next, signal handlers are established to provide for orderly shutdown. The functions `startup()` and `open_hippi()` handle the remaining initialization before processing of input data is begun. An asynchronous read is started on all input buffers before entering the main loop. In the main loop, after an

Table 20 rec_spls source files

File	Short description
rec_spls.c	Main function.
rec_spls.h	General headers and structure definitions.
spl_products.h	Definition of data product structures.
spl_rec_header.h	Definition of header for output data records.
startup.c	Initialization code.
rd_array.c	Function to read array name.
getargs.c	Function to parse command line arguments.
cshm.c	Function to create or attach to shared memory.
open_hippi.c	Function to open HIPPI device.
reset_hippi.c	Function to reset HIPPI device.
hip_set_parms.c	Wrapper for HIPPI <i>ioctl()</i> function.
process_pkt.c	Function to process one packet of data from APP0.
merge_ap.c	Function to merge one aperture's data into output structures.
wrt_fifos.c	Function to write data to output FIFOs.
snd_bips.c	Function to send BIPs to DRP.
snd_sotos.c	Function to send SOTOs to DRP.
wrt_scnb.c	Function to write SCNBs to disk.
end_process.c	End process by performing cleanup and exiting.

aiowait() completes, the data is organized and written to output FIFOs, and a new asynchronous read is started on the buffer processed. Since the asynchronous read operations are not guaranteed to complete in order, a loop is executed to search the buffers for the next frame after each *aiowait()* completes. If data becomes available in all input buffers but the next expected frame is not found, diagnostic information is logged, and the process terminates.

Code in the file *startup.c*, listed in Appendix J, is responsible for most initialization. A file is used as a flag to indicate when this process is ready. In this module, that file is unlinked if it exists at startup. Another flag file that indicates to other processes when data is being stored to disk should not exist at this point and is unlinked if it does. The process detaches and runs as a daemon process if the “-d” option is set. Next, processes are forked that read data from FIFOs in shared memory and write it to the destination. Several data structures are cleared and some other miscellaneous initialization is performed.

The file *rd_array.c*, listed in Appendix K, contains a function that reads the name of the sensor array from a file. The Measurement Facility consists of two Beamformers on which this code runs. The name identifying the Beamformer is set in a file. The path to the name file may be specified by an environment variable. If the name can not be read from the file, a default is used.

The file `getargs.c`, listed in Appendix L, contains a function that processes command line arguments. The `getopt()` function does most of the work.

The file `cshm.c`, listed in Appendix M contains code to create and/or attach to shared memory. The areas specified by `FRAME_SHMKEY` and `STIME_SHMKEY` are used by other processes and usually exist when the process is started.

The file `open_hippi.c`, listed in Appendix N, contains code to open the HIPPI device. Two processes on the node where this code normally runs use the HIPPI device. If another process is using the HIPPI device, a simple open is performed. Otherwise, the device is reset. This code was not included or run from *startup()* so that the main program could be stripped down to only perform the data reads during development.

The code in the file `reset_hippi.c`, listed in Appendix O, resets and opens the HIPPI device. This code executes *ioctl()* functions to perform a reset on the HIPPI board. This driver and board have a tendency to hang and are reset at every opportunity.

The file `hip_set_params.c`, listed in Appendix P, contains a wrapper function for a HIPPI *ioctl()* function, providing an easy method to set channel parameters. This code was adapted from a device diagnostic.

The file `process_pkt.c`, listed in Appendix Q, contains code that extracts data from the input buffers and stores it in the correct destination. Much of this code is devoted to generating headers for the output data streams. One piece of data, the sync time, stays the same during execution of the program. Therefore, logic is included to write this to the header in each FIFO location only once.

The file `merge_ap.c`, listed in Appendix R, contains the function that transfers spectral data from an input buffer to the correct locations in the output FIFOs. This function is called once for each aperture in each frame. For the case where all apertures are in service, this function will be executed five times for each packet received.

On average, every 218 ms a data packet is received from APP0. This packet consists of five structures, one for each aperture. Each structure in turn contains header data, SOTOs, BIPs, custom BIPs, and SCNB data. The spectral products from the individual apertures are combined to form the proper output data packets. This data flow is shown in Figure 27.

`merge_ap` consists of two nearly identical sections. The first section is used for aperture one while the other is used for the other four apertures. The two sections were necessary because the aperture one data structure is larger than the structure for the other apertures. The coarse narrow band data from the input buffer is copied to the destination using the

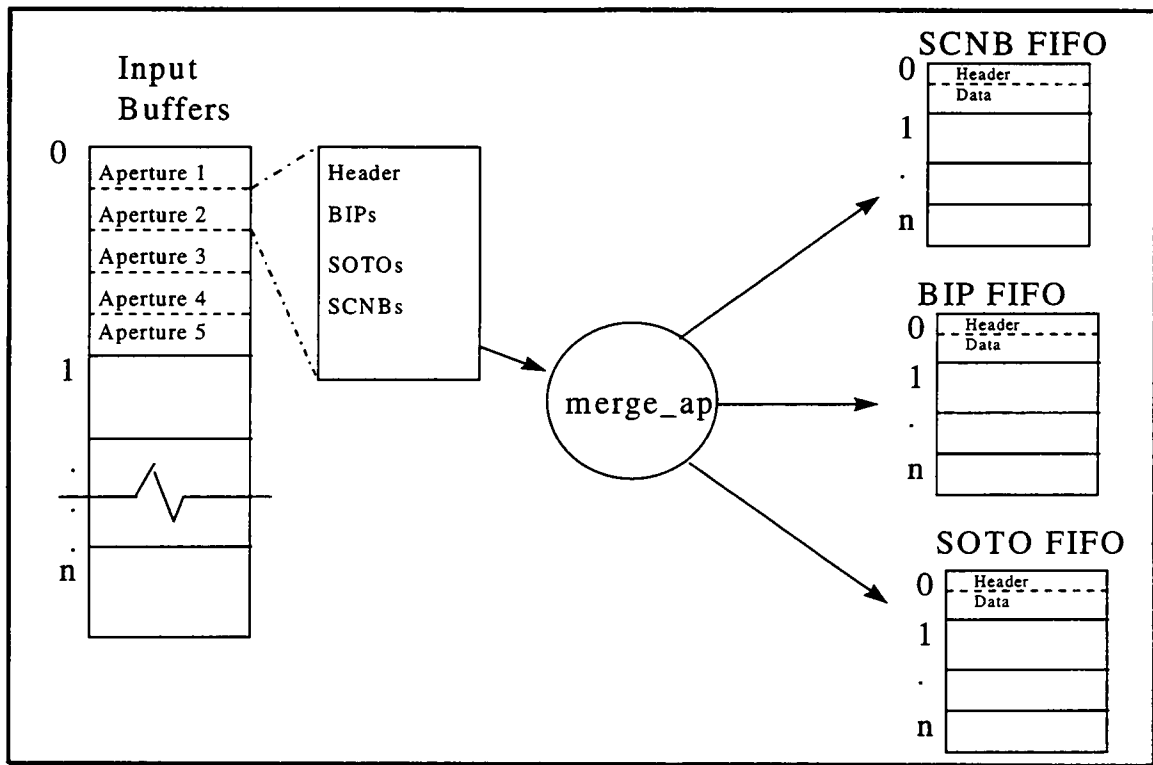


Figure 27 `merge_ap` data flow

memcpy() system function. No rearranging of data is necessary. The BIP data requires more processing. BIPs consist of groups of 35 values. Twenty-five of the values are fixed, but ten custom BIPs may be defined at run time. Each aperture contributes specific one-third octave components of the 25 values in the output. For example, bands zero through twelve come from aperture one, and bands 13, 14 and 15 come from aperture two. Which bands are copied from the input buffer for a given aperture is determined by indexing the *ap_oto_p* array with the aperture number. Custom BIPs are handled differently, because it is not known prior to execution which apertures will contribute a given custom band. To keep this code from needing setup data, all ten custom bands from all apertures are summed. Apertures where custom BIP bands are not computed have zero values for those bands.

The code in the file *wrt_fifos.c*, listed in Appendix S, is used to manage the output FIFOs. The name is somewhat misleading because the data is actually written in *process_pkt()*. Two variables are used with each FIFO, one indicating how full it is (*level*) and the other pointing to where the next data is to be written. The levels are in shared memory and are also written by the processes that read data from the FIFOs. Each time a new frame is received, the levels are incremented and the address pointer is updated. If the child processes are not able to empty the FIFOs, data is overwritten. When overwrites occur, another variable in shared memory is updated that is used by a monitoring process. Messages are logged on buffer full condition and when a full condition ends.

The files `snd_bips.c` and `snd_sotos.c`, listed in Appendix T and Appendix U, respectively, contain functions that read data from the output FIFOs and write to the DRP. The two functions differ only in data source and DRP connection used. During startup, two processes are forked that execute the functions. In other words, these functions loop in independent processes. The functions connect to the shared memory regions and to the DRP before entering the main loop. The main loop executes until the main process writes a “-1” to the FIFO level as a termination flag. The steps executed are

If the FIFO index is greater than zero, write data from the FIFO to DRP, update the FIFO index and decrement the FIFO level.

If the FIFO level is less than zero, exit.

Delay before executing the next loop by executing a *select()* function.

The delay is set for 100 ms so the data writes can keep up with the incoming data and make up for slow writes to DRP.

The file `wrt_scnb.c`, listed in Appendix V, contains code to write SCNB data to disk. The function executes in a process forked from the *main()* `rec_spls` process. Before entering the main loop, this process connects to shared memory regions that contain the output FIFO and control values. Whether or not data is written to disk is controlled by the *store* value in shared memory. This value is written by the storage control process in response to commands from an external system. The main loop checks the value of *store* and the FIFO level every 100 ms. When *store* transitions from zero to one, a new data file is opened for write. If the file becomes full, writes are no longer attempted until *store*

makes another zero to one transition. When data is not being stored to disk, the FIFO index and level are updated.

The file `end_process.c`, listed in Appendix W, contains the process termination code. This function is used for normal termination, for error termination and in response to signals. A delay is first executed to allow time for the output FIFOs to empty. After the delay, a “-1” is written to all the FIFO levels causing child processes to exit. The `wait()` system function is then used to reap the child processes. Next, this process detaches from the shared memory regions and removes the regions not used by other processes. The two flag files used to show when this process is ready and when data is being stored to disk are removed. Finally, the HIPPI device is closed and the process exits. An `ioctl()` function is used to close the HIPPI device. A simple `close()` would not leave the HIPPI device in the correct state.

Conclusion

All problem requirements were met by the data communications design and implementation presented in this thesis. Production use and operational testing have demonstrated that all data communications requirements were satisfied. Measurement results presented in Chapter 6 show a close agreement between expected and actual data transmission times. The Beamformer has run for extended periods without loss of data communications. The design included achieving real-time communications with a non-real-time operating system.

The most important lesson learned from this work is that real-time processing requirements should be included in the procurement of digital systems that must process continuous data streams. Though the system procurement, mentioned in the section on prior work, contains specifications for throughput, real-time characteristics were not specified. This omission led to the problems stated in Chapter 8.

Future research related to topics covered in this thesis should include characterization of the Internet Protocol running over the HIPPI framing protocol, investigation of HIPPI alternatives including Fibre Channel and investigation of DRP alternatives.

List of References

ANSI S1.6-1984, "Preferred Frequencies, Frequency Levels, and Band Numbers for Acoustical Measurements", American National Standards Institute, 1984.

ANSI X3.183-1991, High-Performance Parallel Interface. Mechanical, Electrical, and Signaling Protocol Specification (HIPPI-PH), American National Standards Institute, 1991.

ANSI X3.210-1992, High-Performance Parallel Interface. Framing Protocol (HIPPI-FP) American National Standards Institute, 1992.

ANSI X3.222-1993, High-Performance Parallel Interface. Physical Switch Control (HIPPI-SC) American National Standards Institute, 1993.

IEEE Std 802.2-89, Local Area Networks - Part 2: Logical Link Control, Institute of Electrical and Electronic Engineers, 1989.

ANSI/IEEE Std 802.3-90, Local Area Networks - Part 3: Carrier Sense Multiple Access with Collision Detection, Institute of Electrical and Electronic Engineers, 1990.

Oppenheim, A. V., and R. W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, New Jersey, 1990.

Stern, Hal, *Managing NFS and NIS*, O'Reilly & Associates, Inc., Sebastopol, CA, 1991.

Stevens, W. R., *UNIX Network Programming*, Prentice-Hall, New Jersey, 1990.

Stevens, W. R., *Advanced Programming in the UNIX Environment*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1992.

Tucker, Raymond W., Jr., *Reconstruction of a Composite Time Series Output for a Multiple-Aperture Frequency-Domain Beamformer*, Unpublished Masters Thesis, University of Tennessee, Knoxville, TN, 1995.

Appendices

Appendix A Listing of drp_src.c File

```

/*****
 *
 * Module name:  drp_src.c
 * Date 6/26/95
 *
 * Created by: Bobby R. Whitus
 *
 *****/
 * FILE DESCRIPTION:
 *   This module writes data to DRP.
 *****/

#include <sys/types.h>
#include <stdio.h>
#include <errno.h>
#include <sys/uio.h>
#include <fcntl.h>
#include <malloc.h>
#include <tcpio-sc.h>          /* For DRP */

#define BUGOUT(X) {perror(X); exit(-1);}
#define DEF_DEST_ADD 4        /* Destination address */
#define DEF_LENGTH 1024
#define DEF_LOOP_CNT 100      /* Default number of sends */
#define MAXBUFF 1024*1024
#define NUM_STEPS 8

int      file_flag = 0;          /* Flag to wrt to file */
char     *filename;             /* Filename to wrt to */
int      length = DEF_LENGTH / 4; /* Write length */
int      loop_cnt = DEF_LOOP_CNT; /* Num of reps for timing */

main(argc, argv)
{
    char      *buff;             /* Buffer to hold data */
    FILE      *fp;
    void      getargs();         /* Function to parse command line */
    int       i, j;             /* Indexes */
    double    rate;             /* Data rate */
    double    s;                /* Holds time */
    double    secs();
    double    tsec;             /* Holds accumulated time */
    char      mt1[16];          /* Msg typ bufs, for DRP */
    peer_name_t sndr_name;
    tcpio_status_t status;       /* DRP function status */
    char      peer_name[32];     /* For DRP connect */
    tcpio_status_t drp_status;   /* DRP functions status */
    mssg_type_t mssg_type;       /* For DRP */
    mssg_size_t mssg_sz;        /* For DRP */
    mssg_ptr_t mssg_ptr;        /* For DRP */
    static    lths[] = {8, 32, 128, 512, 2048, 8192, 32768, 131072};

    getargs(argc, argv);
    if ((buff = (char *) malloc(MAXBUFF)) == NULL)
        BUGOUT("drp_src: malloc");
    /*
     * Load simple pattern in output buffer
     */
    for (i = 0; i < length; i++)
        *(buff + i) = i;

    /*
     * Connect to DRP
     */
    (void) strcpy(peer_name, "drp_test_node");

```

```

(void) strcpy(mt1, "drp_test_b");

drp_status = tcpio_drp_connect_va(NULL, peer_name, mt1, "");
if (drp_status != TCPIO_OK)
{
    printf("Error on tcp_drp_connect\n");
    tcpio_print_status(drp_status);
    exit(-1);
}

/*
 * Write and read data
 */
for (i = 0; i < NUM_STEPS; i++)
{
    tsec = 0; /* Accumulate total secs for all reps */
    for (j = 1; j <= loop_cnt; j++)
    {
        s = secs(); /* Get start time */
        /* Send the message using DRP */
        status = tcpio_send_mssg("drp_test_a", lths[i], buff,
                                (tcpio_flag_t) BLOCKING_IO);
        if (status != TCPIO_OK)
        {
            (void) printf("drp_src: drp send error");
            /* Print status of DR send */
            tcpio_print_status(status);
        }
        /* Receive DRP message using sync. Read */
        status = tcpio_rcv_mssg(sndr_name, mssg_type, &mssg_sz,
                                &mssg_ptr, BLOCKING_IO);
        if (status != TCPIO_OK)
        {
            tcpio_print_status(status);
            if (status == TCPIO_RESYNCHED)
                printf("acknowledging resynch...\n");
            else
                exit(-1);
        }
        else if (mssg_sz != lths[i])
            printf("\nIncorrect message length on receive\n");
        /* free buffer returned by tcpio_rcv_mssg() */
        (void) tcpio_free(mssg_ptr, __FILE__, __LINE__);
        memcpy(buff, mssg_ptr, mssg_sz);

        /* free buffer returned by tcpio_rcv_mssg() */
        (void) tcpio_free(mssg_ptr, __FILE__, __LINE__);

        s = secs() - s;
        tsec += s; /* Find total time */
    }
    /* Calculate data rate */
    rate = (loop_cnt * lths[i] * 2 / (1000.0 * tsec));
    printf("\nlength\t%i\ttrate\t%8.3f KBs", lths[i], rate);
    printf("\n\n");

    if (file_flag)
    {
        if ((fp = fopen(filename, "a")) == (FILE *) NULL)
            BUGOUT("app_src: fopen");
        fprintf(fp, "\n%i\t\t%8.3f", lths[i], rate);
        fclose(fp);
    }
}
/*

```

```

    * Close DRP connection
    */
    (void) tcpio_drp_disconnect();

    exit(-1);
}
/*
=====
* FUNCTION secs - Get seconds for timing.
*
*-----
*/
#include <sys/time.h>

double
secs()
{
    struct timeval ru;

    gettimeofday(&ru, (struct timezone *) 0);
    return (ru.tv_sec + ((double) ru.tv_usec) / 1000000);
}
/*
=====
* FUNCTION getargs - Processes command line arguments
*
*-----
*/
void
getargs(argc, argv)
    int      argc;
    char     *argv[];
{
    extern char *optarg;      /* Used by getopt stuff */
    extern int  optind;       /* Used by getopt stuff */
    extern int  opterr;       /* Used by getopt stuff */
    int        c;
    int        errflg = 0;    /* Error flag */
    /* Loop using getopt function, this does most of the work */
    while ((c = getopt(argc, argv, "hl:c:f:")) != -1)
        switch (c)
        {
            case 'l':
                length = (atoi(optarg));      /* Convert ASCII to int */
                break;
            case 'c':
                loop_cnt = atoi(optarg);        /* Convert ASCII to int */
                break;
            case 'h':
                errflg++;
                break;
            case 'f':
                file_flag = 1;
                filename = optarg;
                break;
            case '?':
                errflg++;
        }
    if (errflg)
    {
        (void) printf("\nUsage app_src [-d dest] [-l length] [-c count] [-r] \n");
    }
    printf("\ndrp_src");
    printf("\nWrite length(Bytes): %i", length * 4);
}

```

```
    return;  
}  
/*=====*/
```

Appendix B Listing of drp_dest.c File

```

/*****
*
* Module name:  drp_dest.c
* Date 6/26/95
*
* Created by: Bobby R. Whitus
*
*****/
* FILE DESCRIPTION:
*This module echos DRP messages.
*****/

#define BUGOUT(X) {perror(X); exit(-1);}
#define MAX_LENGTH 1024*1024*4
#include <stdio.h>
#include <unistd.h>
#include <netinet/in.h>
#include <signal.h>
#include <tcpio-sc.h>

void          sigint_handler();
int           total_rec;          /* Total DRP messages received */

int
main(argc, argv)
    int       argc;
    char      **argv;
{
    tcpio_status_t  status;
    int             tmp;
    int             n = 1;
    char            *buff;
    int             i, j, k, l;
    int             length=MAX_LENGTH;
    char            junk[16];
    peer_name_t     sndr_name;
    mssg_type_t     mssg_type;
    mssg_size_t     mssg_sz;
    mssg_ptr_t      mssg_ptr;
    static mssg_type_t mssg_types[] = {"drp_test_a",
    ""};
    /* trap CTRL-C interrupts */
    signal(SIGINT, sigint_handler);
    total_rec = 0;

    /* ,alloc space for receive buffer */
    if ((buff = (char *) malloc(length)) == NULL)
        BUGOUT("drp_dest: malloc");
    /* Connect to DRP specifying string of message types */
    status = tcpio_drp_connect(argv[1], "drp_dest", mssg_types);
    if (status != TCPIO_OK)
    {
        tcpio_print_status(status);
        exit(-1);
    }

    for (;;)
    {
        /* Receive DRP message */
        status = tcpio_rcv_mssg(sndr_name, mssg_type, &mssg_sz,
                                &mssg_ptr, BLOCKING_IO);
        if (status != TCPIO_OK)

```

```

    {
        tcpio_print_status(status);
        if (status == TCPIO_RESYNCHED)
            printf("acknowledging resynch...\n");
        else
            exit(-1);
    }
    else /* If receive was OK, send back */
    {
        total_rec++;
        memcpy(buff, mssg_ptr, mssg_sz);
        status = tcpio_send_mssg("drp_test_b", mssg_sz, buff,
                                (tcpio_flag_t) BLOCKING_IO);
        if (status != TCPIO_OK)
        {
            (void) printf("drp_dest: drp send error");
            tcpio_print_status(status);
        }

        /* free buffer returned by tcpio_recv_mssg() */
        (void) tcpio_free(mssg_ptr, __FILE__, __LINE__);
    }

}

}
/*-----*/
void
sigint_handler()
{
    printf("inside sigint_handler...\n");
    (void) signal(SIGINT, SIG_IGN);
    (void) tcpio_drp_disconnect();
    printf("\nTotal DRP messages received = %i\n", total_rec);
    exit(0);
}
/*=====*/

```

Appendix C Listing of open_ts_sun Function

```

/*
 * =====
 * FUNCTION
 *   open_ts_sun - Open Hippi I/O channel for TS to Sun
 *
 * SYNOPSIS
 *   int open_ts_sun()
 *
 * DESCRIPTION
 *   open_ts_sun_ opens the Hippi device defined by SPL_STEER PATH
 *   for read and write access. The open is done on splsfid which is
 *   used to send spectral products to the Sun. APP0 will also
 *   use this fd to receive steering data from a Sun. The Hippi
 *   channel must be configured to use the framing protocol. This
 *   function blocks until completion and is normally used only on APP0.
 *   This function is FORTRAN callable.
 *
 * RETURN VALUES
 *   open_ts_sun() returns:
 *
 *   nonnegative   on success.
 *   -1           on failure and writes an error message to stderr.
 * -----
 */
int
open_ts_sun()
{
    if ((ts2sfd = open(TS_APP0_SUN_PATH, O_WRONLY)) == -1)
    {
        perror("open_ts_sun: Open error ");
        return -1;
    }

    ifield = TS_DEST_ADD | 0x01000000; /* Set camp-on bit */
    if (ioctl(ts2sfd, HPSC_SETIFIELD, &ifield) < 0)
    {
        perror("open_ts_sun: ioctl error");
        return -1;
    }
    tscnt = 0; /* Remove after test */

    return ts2sfd;
}

```

Appendix D Listing of send_spls Function

```

/*
 * =====
 * FUNCTIONS
 *   send_spls_ - Write beam data via Hippi to Host
 *
 * SYNOPSIS
 *   int send_spls_(spl_products, spl_products_len)
 *   char  *spl_products;
 *   int    *spl_products_len;
 *
 * DESCRIPTION
 *   This function is a FORTRAN callable wrapper for a
 *   awrite used to pass a buffer containing all the SPL products
 *   to the Host. The file descriptor is static to this file.
 *   open_apl and close_spl must be used
 *   to open and close the Hippi channel.
 *   The await_ should be used to determine when the I/O has completed.
 *
 * RETURN VALUES
 *   On success the function returns the request block number for the
 *   awrite performed. -1 is returned if an error occurs.
 * -----
 */
int
send_spls_(spl_products, spl_products_len)
char      *spl_products;
int        *spl_products_len;
{
    static int    cttotal = 0;    /* Transmission count used for testing */
    int          rb;             /* Write request block */

    /*
     * For debugging only
     */
    if (pmilestones)
    {
        splcnt++;
        if (splcnt == 50)
        {
            cttotal += splcnt;
            (void) printf("\nsend_spls: SPL packets sent = %i", cttotal);
            splcnt = 0;
        }
    }
    if ((rb = awrite(splfd, spl_products, *spl_products_len)) < 0)
    {
        perror("send_spls: awrite error");
        return -1;
    }
    return rb;
}

```

Appendix E Listing of await Function

```

/*
* =====
* FUNCTION
*   await_ - Wait for io to complete
*
* SYNOPSIS
*   int wait_(request_id, wait)
*   int request_id;
*   int wait;
*
* DESCRIPTION
*   This is a FORTRAN wrapper for Cray's await function. Se the
*   man page for await for details.
*
* PARAMETERS
*   request_id
*       Specifies the ID value returned by the call that
*       started the request.
*
*   wait
*       When this value is 1, it specifies await should
*       block until the request completes or a signal
*
* RETURNS
*   If wait_ has a value of 1, io_await blocks until the request
*   completes or a signal occurs. If a signal occurs, await_
*   returns -1 and sets errno to EINTR. If wait is zero and the
*   request is not complete, await_ returns -1 and sets errno
*   to EWOULDBLOCK. Note, the request is still active and
*   requires another call to await_ in either of the cases
*   above. Otherwise, the return value is determined by the
*   operation that initiated the request.
* -----
*/
int
await_(rb, wait)
    int      *rb;
    int      *wait;
{
    int      status;
    if ((status = await(*rb, *wait)) < 0)
        if(*wait != 0) /* -1 is a normal return value for non-wait (0) mode */
            perror("await err");
    return status;
}

```

Appendix F Listing of rec_spls.h File

```

/* "@(#)rec_spls.h 1.11" */
/*****
*
* ORNL I&C Division
* Module name:  rec_spls.h
* Version: 1.11
* Modified: 6/26/95 10:21:23
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
*             Oak Ridge National Laboratory
*             Instrumentation & Controls Division
*             Real Time Computer Systems Group
*             P O Box 2008, Bldg 3500, MS 6007
*             Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FILE DESCRIPTION:
*
* Include file for rec_spls main and functions.
*
* Contains include files, definitions, structure definitions, global
* variable definitions used by all rec_spls functions. Some macro
* definitions are in specific functions if they are not used globally.
*
* The main function must define MAIN
*
*****/
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/file.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/asynch.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <malloc.h>
#include <signal.h>
#include <memory.h>
#include <malloc.h>
#include <termios.h>
#include "hipd.h"
#include <bftime.h>
#include <bftmsg.h>
#include "spl_products.h"
#include <spl/spl_rec_header.h>
#include <tcpio-sc.h>
#include <netinet/in.h>

#define A1CNBLEN 884 * NUM_BEAMS * 4 /* Length of aperture 1 CNB beam data */
#define A25CNBLEN 442 * NUM_BEAMS * 4 /* Length of aperture 2-5 CNB beam
* data */
#define BIP_DRP_FIFO_LEN 40 /* BIP DRP write FIFO length */
#define FIRST_AIO_TIMEOUT 40 /* Timeout for first Hippi aiowait (secs) */
#define HIP_PATH "/dev/hipi0" /* Hippi device */
#define INPUT_BUFNUM 10 /* Number of input buffers */
#define RUN_AIO_TIMEOUT 1 /* Timeout for most Hippi aiowaits (secs) */
#define SCNB_DSK_FIFO_LEN 40 /* STI CNB disk write FIFO length */
#define SOTO_DRP_FIFO_LEN 40 /* OTO DRP write FIFO length */

```

```

/* The following define which OTOs come from which aperture */
#define AP1_OTO_START      0
#define AP1_OTO_END      12
#define AP2_OTO_START      13
#define AP2_OTO_END      15
#define AP3_OTO_START      16
#define AP3_OTO_END      18
#define AP4_OTO_START      19
#define AP4_OTO_END      21
#define AP5_OTO_START      22
#define AP5_OTO_END      24

/* The following are for shared memory areas */
#define SCNB_SHMKEY 1000
#define SOTO_SHMKEY 1001
#define BIP_SHMKEY 1002
#define CTL_SHMKEY 1003
#define FRAME_SHMKEY 3000 /* Frame shared memory */
#define STIME_SHMKEY 2000 /* Sync time shared memory */
#define STORE_SHMKEY 4000 /* STI OTO Store flag shared memory */
#define SHM_PERMS 0666 /* Permissions */
#define SCNB_WRT_SLEEP 50000 /* Poll time for STI CNB disk write process */
#define SOTO_WRT_SLEEP 50000 /* Poll time for STI OTO DRP write process */
#define BIP_WRT_SLEEP 50000 /* Poll time for BIP DRP write process, (us) */
/*

static char      SccsID_rec_spls_h[] = "@(#)rec_spls.h1.11"; /* For ver control
*/

/*
 * Local structure definitions
 */
typedef struct
{
    int          start;
    int          end;
}                aperture_oto_positions_t;

typedef struct                /* For BIP oto wrts to DRP */
{
    struct spl_rec_header_rec head;
    float          bip_otos[NBIPS][35];
}                bip_oto_rec_t;

typedef struct
{
    int          scnb;
    int          soto;
    int          bip;
    int          scnbs_lost; /* The "_lost" variables were added */
    int          sotos_lost; /* for monitoring by track_monitor */
    int          bips_lost; /* (RC 11/17/94) */
}                fifo_level_t;

typedef struct                /* For STI oto wrts to disk and DRP */
{
    struct spl_rec_header_rec head;
    float          sti_otos[NUM_HBEAMS][NUM_VBEAMS][25];
}                sti_oto_rec_t;

typedef struct                /* Time and time ready flag in shared mem */
{
    bftime_t      bft; /* Beamformer time structure */

```

```

    int                track_pid;    /* Set when other process has written bftime
*/
}                stime_t;    /* Short for shared time */

typedef struct
{
    spl_products_a1_t  a1;
    spl_products_a2_t  a2_5[5];    /* Extra aperture space used for
                                   * mysterious overwrites */
}    inbuf_t;

/*
 * Global variable definitions and declarations
 */

#ifdef MAIN

struct aio_result_t aio_result[INPUT_BUFNUM]; /* Async I/O struct */
int                bc_1st_act_aper;    /* BC 1st active aperture */
char                bfname[4]; /* Beamformer ID */
char                *bfname_p = &bfname[0]; /* Pointer to bfname */
char                *bips;    /* Pointer to BIP shared memory */
int                bip_fifo_idx = 0; /* BIP FIFO index, indexes into fifo */
int                bip_pid;    /* BIP write process pid */
bip_oto_rec_t      *bip_rec;    /* BIP record */
int                bipsmid;    /* BIP shared memory id */
int                buf_ms = 0;    /* Buffer milestone log interval */
int                ctlsmid;    /* FIFO control shared memory id */
extern int         debug;
int                detach = 0;    /* Create detached process ie. daemonize */
int                exp_rcv_flags;
fifo_level_t      *fifo_level;    /* Fifo levels */
int                *frame;    /* Pointer to frame number in shared mem */
int                frame_smid;    /* Shared memory id for frame number */
int                hfd;
inbuf_t           inbuf[INPUT_BUFNUM]; /* Receives data packets from APP0 */
int                no_bips2drp = 0; /* Inhibit writing BIPs to DRP */
int                no_otos2drp = 0; /* Inhibit writing OTO STIs to DRP */
int                no_stis2dsk = 0; /* Inhibit writing CNB STIs to disk */
int                num_children = 0; /* Number of child processes */
char                rdy_path[64];    /* Path for rec_spls_rdy file */
char                *scnb;    /* temp storage used to merge beams for STI
                                   * CNB wrt */

extern struct schmid_ds;
int                scnb_cpy_idxs[5]; /* Indexes for STI CNB copies */
int                scnb_fifo_idx = 0; /* STI CNB FIFO index, indexes into
                                   * fifo */
int                scnb_pid; /* STI CNB write process pid */
int                scnb_wrt_len; /* Length for combined header-all aper buffer
                                   * used for combining data and performing sti
                                   * cnb wrt */
int                scnbsmid; /* STI CNB shared memory id */
int                sf_1st_act_aper; /* SF 1st active aperture */
struct spl_rec_header_rec shead; /* spl header */
int                soto_fifo_idx = 0; /* STI OTO FIFO index, indexes into
                                   * fifo */
int                soto_pid; /* STI OTO write process pid */
int                sotosmid; /* DTI OTO shared memory id */
char                *sotos;    /* Pointer to SOTO shared memory */
char                sstis_path[64]; /* Path for STORING_STISIS file */
sti_oto_rec_t      *sti_oto_rec;
stime_t           *stime;    /* Pointer to sync time shared memory */
int                stime_smid; /* Shared memory sync time id */

```

```

int          store_smid;          /* Shared memory store flag memory id */
int          *store;              /* Shared memory flag to store STI CNB data
                                  * to file */

#else

extern struct aio_result_t aio_result[INPUT_BUFNUM]; /* Async I/O struct */
extern int    bc_1st_act_aper;    /* BC 1st active aperture */
extern char   *bips;              /* Pointer to BIP shared memory */
extern int    bip_fifo_idx;
extern int    bip_pid;            /* BIP write process pid */
extern bip_oto_rec_t *bip_rec;    /* BIP record */
extern int    bipsmid;            /* BIP shared memory id */
extern char   *bfname_p;
extern int    buf_ms;             /* Buffer milestone log interval */
extern int    ctlsmid;            /* FIFO control shared memory id */
extern int    debug;
extern int    detach;             /* Create detached process ie. daemonize */
extern int    exp_rcv_flags;
extern fifo_level_t *fifo_level; /* Fifo levels */
extern int    *frame;             /* Pointer to frame num in shared mem */
extern int    frame_smid;         /* Shared memory id for frame number */
extern int    hfd;
extern inbuf_t inbuf[];
extern int    no_bips2drp;        /* Inhibit writing BIPs to DRP */
extern int    no_otos2drp;        /* Inhibit writing OTO STIs to DRP */
extern int    no_stis2dsk;        /* Inhibit writing CNB STIs to disk */
extern int    num_children;       /* Number of child processes */
extern char   rdy_path[];         /* Path for rec_spls_rdy file */
extern struct schmid_ds;
extern char   *scnb;              /* temp storage used to merge beams for STI
                                  * CNB wrt */
extern int    scnb_cpy_idxs[];    /* Indexes for STI CNB copies */
extern int    scnb_pid;           /* STI CNB write process pid */
extern int    scnb_wrt_len;
extern int    scnb_fifo_idx;      /* STI CNB FIFO index, indexes into fifo */
extern int    scnb_smid;          /* STI CNB shared memory id */
extern int    sf_1st_act_aper;    /* SF 1st active aperture */
extern struct spl_rec_header_rec shead; /* spl header */
extern int    soto_fifo_idx;      /* STI OTO FIFO index, indexes into fifo */
extern int    soto_pid;           /* STI OTO write process pid */
extern int    sotosmid;           /* DTI OTO shared memory id */
extern char   *sotos;             /* Pointer to SOTO shared memory */
extern char   sstis_path[];       /* Path for STORING_STISIS file */
extern sti_oto_rec_t *sti_oto_rec;
extern time_t *stime;             /* Pointer to sync time shared memory */
extern int    stime_smid;         /* Shared memory sync time id */
extern int    store_smid;         /* Shared memory store flag memory id */
extern int    *store;             /* Shared memory flag to store STI CNB data
                                  * to file */

#endif

/*=====*/

```

Appendix G Listing of spl_products.h File

```

/* "@(#)spl_products.h      1.6" */
/*****
*
*                               ORNL I&C Division
* Module name:  spl_products.h
* Version: 1.6
* Modified: 10/4/95 12:38:25
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
*           Oak Ridge National Laboratory
*           Instrumentation & Controls Division
*           Real Time Computer Systems Group
*           P O Box 2008, Bldg 3500, MS 6007
*           Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FILE DESCRIPTION
*   Include file that defines the structure that contains the beam
*   products for each frame for one aperture.
*****/
#define NOTOS 25
#define NCOTOS 10
#define NUM_BEAMS 12
#define NUM_HBEAMS 6
#define NUM_VBEAMS 2
#define NBIPS 348
#define NOTOS_A1 13
#define NOTOS_A2 3
#define NBINS_A1 884
#define NBINS_A2 442

static char  SccsID_spl_products_h[] = "@(#)spl_products.h      1.6"; /* For
ver control */

/*
* NOTE: The size of the following structures must be a multiple of 16 bytes
*/
struct sti_oto_head_rec
{
    int          aperture;
    int          buf_num;
    int          sti_size;
    int          num_sti_otos;
    int          num_bip_otos;
    int          store;
    int          last_record; /* Tell the APP to die */
    int          spare;
};

struct steering_rec
{
    float         range;
    float         azimuth;
    float         dep_elev;
    char          pad[4]; /* So struct is a multiple of 16 bytes */
};

struct beam_steer_rec
{
    float         measr_azim;          /* Measurement beam arimuth */
    float         measr_elev;          /* Measurement beam elevation */
    float         image_azim;          /* Image beam azimuth */
    float         image_elev;          /* Image beam elevation */
    float         measr_hoff[5];       /* Measurement beam horz offsets */

```

```

float      measr_voff[2];    /* Measurement beam vert offsets */
char       pad[4];
};

typedef struct
{
    struct sti_oto_head_rec head;
    struct steering_rec steer;
    struct beam_steer_rec beam_steer;
    float    scnb[NUM_HBEAMS][NUM_VBEAMS][NBINS_A1]; /* sti cnb (beams) */
    float    sotos[NUM_HBEAMS][NUM_VBEAMS][NOTOS_A1]; /* sti otos */
    float    botos[NBIPS][NOTOS_A1];                /* bip otos */
    float    bcotos[NBIPS][NCOTOS];                  /* bip custom otos */
}    spl_products_a1_t;

typedef struct
{
    struct sti_oto_head_rec head;
    struct steering_rec steer;
    struct beam_steer_rec beam_steer;
    float    scnb[NUM_HBEAMS][NUM_VBEAMS][NBINS_A2]; /* sti cnb (beams) */
    float    sotos[NUM_HBEAMS][NUM_VBEAMS][NOTOS_A2]; /* sti otos */
    float    botos[NBIPS][NOTOS_A2];                /* bip otos */
    float    bcotos[NBIPS][NCOTOS];                  /* bip custom otos */
}    spl_products_a2_t;

```

Appendix H Listing of spl_rec_header.h File

```

/* "(@#)spl_rec_header.h 1.4" */
/*****
*
* ORNL I&C Division
* Module name: spl_rec_header.h
* Version: 1.4
* Modified: 10/4/95 12:37:54
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
* Oak Ridge National Laboratory
* Instrumentation & Controls Division
* Real Time Computer Systems Group
* P O Box 2008, Bldg 3500, MS 6007
* Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FILE DESCRIPTION
* Include file that defines the file/comm
* header for SPL data written to drp and to file.
*****/
#ifndef _spl_rec_header_h
#define _spl_rec_header_h

#ifndef _bftime_h /* Get bftime def if not already done */
#include <bftime.h>
#endif

static char SccsID_spl_rec_header_h[] = "(@#)spl_rec_header.h 1.4 10/4/95
ORNL";/* For ver control*/

struct spl_rec_header_rec
{
    char          array[8]; /* Array name (ex. Hayes) */
    bftime_t      time; /* Last sync time */
    int           buf_num; /* Buffer number */
    float         sf_range; /* SF range measurement */
    float         sf_azimuth; /* SF azimuth angle */
    float         sf_dep_elev; /* SF depression elevation angle*/
    char          resv1[72]; /* Padding */
    int           apex; /* Apertures expected */
    int           apval; /* Apertures valid */
    float         bc_range; /* BC range measurement */
    float         bc_azimuth; /* BC azimuth angle */
    float         bc_dep_elev; /* BC depression elevation angle*/
    float         sf_measr_azim; /* SF measurment beam azimuth */
    float         sf_measr_elev; /* SF measurement beam elevation */
    float         sf_image_azim; /* SF image beam azimuth */
    float         sf_image_elev; /* SF image elevation */
    float         sf_measr_hoff[5]; /* SF measr beam horz offsets */
    float         sf_measr_voff[2]; /* SF measr beam vert offsets */
    float         bc_measr_azim; /* BC measurment beam azimuth */
    float         bc_measr_elev; /* BC measurement beam elevation */
    float         bc_image_azim; /* BC image beam azimuth */
    float         bc_image_elev; /* BC image elevation */
    float         bc_measr_hoff[5]; /* BC measr beam horz offsets */
    float         bc_measr_voff[2]; /* BC measr beam vert offsets */
    char          resv2[120]; /* Padding */
};

#endif /* !_spl_rec_header_h*/

```

Appendix I Listing of rec_spls.c File

```

/*****
*
*                               ORNL I&C Division
* Module name:  rec_spls.c/* "@(#)rec_spls.c 1.13" */
*
* Version: 1.13
* Modified: 6/21/95 16:38:25
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
*           Oak Ridge National Laboratory
*           Instrumentation & Controls Division
*           Real Time Computer Systems Group
*           P O Box 2008, Bldg 3500, MS 6007
*           Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FILE DESCRIPTION:
*   This file contains functions to receive beam STIs, OTOs and image
*   OTOs from the APPs.
*
*   Usage  rec_spls [-d] [-s] [-t] [-b] [-a mask]
*           -d  detach process
*           -s  inhibit STI writes to disk
*           -o  inhibit OTO writes to drp
*           -b  inhibit BIP writes to drp
*           -a x specify active aperture mask, default is 31 for all
*           -m x specify milestone be printed every x packets received.
*
* INPUTS
*   Each aperture sends a packet of data over the Hippi link to this
*   process. Each packet contains Short Term Integrated Course Narrow
*   Band data (STI CNBs), STI One Third Octave data (OTOs) and Beam
*   Integral Products (BIPs). The data packet also contains header
*   information including steering data used to make the data and
*   a flag used to indicate the last record from the aperture. Each
*   aperture sends a packet every 218ms. The packet data structure
*   is defined in spl_products.h.
*
*   The array name is read from a path specified by the environment
*   variable ARRAY_PATH. If this variable is not set, the path
*   defined by DEF_ARRAY_PATH is used.
*
*   The sync time is read from a path specified by the environment
*   variable TIME_PATH. If this variable is not set, the path
*   defined by DEF_TIME_PATH is used.
*
* OUTPUTS
*   The data from the 5 apertures is merged for each 218ms frame and
*   output to three destinations. STI CNB data is written to a local
*   disk. STI CNB data is written to the path specified by the
*   STI_CNB_PATH environment variable or to the path defined by
*   DEF_STI_CNB_PATH if the environment variable is not set.
*   STI OTO and BIP data is sent to other networked nodes using
*   the Data Routing Process (DRP). All output data records include a
*   reader defines in spl_rec_header.h.
*
*****/
#define MAIN /* For use in include file */
#include "rec_spls.h"

int
main(argc, argv)
    int      argc;

```

```

    char            *argv[];
{
    static char      SccsID[] = "@(#)rec_spls.c 1.13 6/21/95 ORNL"; /* For ver
control */
    extern aio_result_t *aiowait();
    struct aio_result_t *aio_result_p; /* For aiowait return */
    int                buf; /* Used to loop through input buffers */
    int                cbuf; /* Current buffer */
    void               end_process();
    int                hip_rd_len; /* Length for HIPPI reads */
    void               open_hippi();
    int                n;
    int                i; /* index */
    int                least_frame; /* The smallest frame number in the input
array
                                * that has not been processed yet. */
    int                least_frame_buf; /* The buffer number contained in
                                * buffer specified by least_frame */
    void               process_pkt();
    int                no_least_frame; /* If 1, no least frame has been
                                * chosen yet. */
    int                rec_cnt = 0; /* Packets received */
    void               startup();
    struct timeval     timeout; /* Time struct for read timeout */
    void               wrt_fifos();

    pname = argv[0]; /* Set for error messages */
    log_init(); /* Set up message nad error logging */
    exp_rcv_flags = 31; /* Look for all apertures as default */
    debug = 1;

    /* Calculate size of HIPPI reads */
    hip_rd_len = sizeof(spl_products_a1_t) + 4 * sizeof(spl_products_a2_t);
    printf("hip_rd_len: %d\n", hip_rd_len);

    (void) printf("\nEXECUTING rec_spls version 1.13, modified 6/21/95 16:38:25
\n");
    /*
    * Set up signal catchers
    */
    if (((int) signal(SIGINT, end_process)) < 0)
    {
        perror("Signal function error, SIGINT\n");
    }

    if (((int) signal(SIGTERM, end_process)) < 0)
    {
        perror("Signal function error, SIGTERM\n");
    }

    /*
    * Initialize everything
    */
    startup(argc, argv);

    /*
    * Open Hippi device Hippi is not opened in startup to allow program to
    * be modified to just be a data sink for the SPL writes.
    */
    open_hippi(); /* Open and init Hippi link */
    /*
    * Start a read on all buffers
    */

```

```

for (i = 0; i < INPUT_BUFNUM; i++)
{
    /* Set return values to AIO_INPROGRESS */
    aio_result[i].aio_return = AIO_INPROGRESS;

    /* Issue asynchronous read */
    n = aioread(hfd, (&inbuf[i]), hip_rd_len, 0, SEEK_SET, &aio_result[i]);
    /* Check asynchronous read */
    if (n < 0)
    {
        logerr("aioread error ");
    }
}

cbuf = 0;                /* Start at the first buffer */

/*
 * Set first timeout long so process will not exit before SPLs start being
 * sent. Then after first wait use a short time so process will die if
 * data stops
 */
timeout.tv_sec = FIRST_AIO_TIMEOUT;
timeout.tv_usec = 0;      /* Set timeout for I/O */
aio_result_p = aiowait(&timeout); /* Wait for first Hippi I/O */
timeout.tv_sec = RUN_AIO_TIMEOUT;
timeout.tv_usec = 0;      /* Set timeout for non-first I/O */

for (;;)
{
    if ((int) aio_result_p <= 0)
    {
        logmsg("Timeout occurred");
        end_process();
    }
    /* Find smallest frame number of the buffers which have been received */
    no_least_frame = 1;
    for (buf = 0; buf < INPUT_BUFNUM; buf++)
        if (aio_result[buf].aio_return != AIO_INPROGRESS)
            if (no_least_frame ||
                inbuf[buf].a1.head.buf_num < least_frame)
            {
                no_least_frame = 0;
                least_frame = inbuf[buf].a1.head.buf_num;
                least_frame_buf = buf;

                if (least_frame == 0)
                {
                    logmsg("aperture %i", inbuf[buf].a1.head.aperture);
                    logmsg("buf_num %i", inbuf[buf].a1.head.buf_num);
                    logmsg("sti_size %i", inbuf[buf].a1.head.sti_size);
                    logmsg("num_sti_otos %i",
                        inbuf[buf].a1.head.num_sti_otos);
                    logmsg("num_bip_otos %i",
                        inbuf[buf].a1.head.num_bip_otos);
                }
            }

    /* report error if all async operations are in process */
    if (no_least_frame)
    {
        logmsg("Error: aiowait completed with no new frame received");
        end_process();
    }
}

```

```

    }

    /* the buffer which received the least buffer becomes current buffer */
    cbuf = least_frame_buf;

    /* report error if return value is incorrect */
    if (aio_result[cbuf].aio_return != hip_rd_len)
    {
        errno = aio_result[cbuf].aio_errno;
        logerr("Read error");
        end_process();
    }

    /* Organize and output data */
    process_pkt(cbuf);

    /* Set return value to be AIO_INPROGRESS */
    aio_result[cbuf].aio_return = AIO_INPROGRESS;

    /* Issue new read for current buffer */
    n = aioread(hfd, (&inbuf[cbuf]), hip_rd_len,
        0, SEEK_SET, &aio_result[cbuf]);

    /* Check asynchronous read */
    if (n < 0)
    {
        logerr("aioread error ");
        end_process();
    }
    cbuf = (++cbuf) % INPUT_BUFNUM;
    aio_result_p = aiowait(&timeout); /* Wait for next Hippi I/O */
    if (buf_ms)
    {
        if (((++rec_cnt) % buf_ms) == 0)
        {
            logmsg("%i SPL packets received from APP", rec_cnt);
        }
    }
}

}

/*=====*/

```

Appendix J Listing of startup.c File

```

/* "(@"#)startup.c 1.6" */
/*****
*
* ORNL I&C Division
* Module name: startup.c
* Version: 1.6
* Modified: 2/16/95 12:39:22
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
* Oak Ridge National Laboratory
* Instrumentation & Controls Division
* Real Time Computer Systems Group
* P O Box 2008, Bldg 3500, MS 6007
* Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FUNCTIONAL DESCRIPTION:
* Performs most initialization for rec_spls
*
*
* STATUS RETURN VALUES
* NONE
*
*****/
#include "rec_spls.h"

#define A1BIT 1
#define A2BIT 2
#define A3BIT 4
#define A4BIT 8
#define A5BIT 16

#define DEF_REC_SPLS_RDY_PATH "/oper/etc/REC_SPLS_RDY"
#define DEF_STORING_STIS_PATH "/oper/etc/STORING_STIS"

void
startup(argc, argv)
int
argc;
char
*argv[];

{
    static char
        SccsID[] = "(@"#)startup.c 1.6 2/16/95 ORNL"; /* For ver
control */

    static
        ap_bits[] = {A1BIT, A2BIT, A3BIT, A4BIT, A5BIT};
    void
        cshm();
    char
        *epathp; /* Pointer to paths from env */
    int
        fd; /* File descriptor */
    void
        getargs();
    int
        i; /* Index */
    int
        n;

    void
        rd_array();
    void
        snd_bips();
    void
        snd_scnb();
    void
        snd_sotos();
    char
        *sret;
    void
        wrt_scnb();

    /*
     * Get path to file used to tell others that rec_spls is ready
     */
    rdy_path[0] = 0; /* Null terminate string */

    if ((epathp = getenv("REC_SPLS_RDY_PATH")) != NULL)

```

```

    sret = strcpy(rdy_path, epathp);
else
    sret = strcpy(rdy_path, DEF_REC_SPLS_RDY_PATH);

/*
 * Unlink rec_spls flag file if it exist ( should not be there )
 */
(void) unlink(rdy_path);

/*
 * Get path to file that tells integration proc. data is being stored
 */
sstis_path[0] = 0;          /* Null terminate string */

if ((epathp = getenv("STORING_STIS_PATH")) != NULL)
    sret = strcpy(sstis_path, epathp);
else
    sret = strcpy(sstis_path, DEF_STORING_STIS_PATH);

/*
 * Unlink storing file if it exist
 */
(void) unlink(sstis_path);

/*
 * Get command line options
 */
getargs(argc, argv);

/*
 * Get ID of this beamformer
 */
if (get_bf_id(bfname_p) < 0)
{
    logmsg("Error getting beamformer id");
    exit(-1);
}

/*
 * If desired, detach process
 */
if (detach)
{
    if ((i = fork()) < 0)
    {
        logerr("Fork error");
        exit(-1);
    }
    else if (i > 0)
    {
        /* Parent */
        (void) printf("\nChild pid = %i, Parent exiting...\n", i);
        exit(0);
    }
    else
    {
        debug = 0;          /* Sends messages to Log */
        n = getdtablesize(); /* See how many fds */

        if ((fd = open("/dev/tty", O_RDWR)) >= 0)
        {
            if ((ioctl(fd, TIOCNOTTY, (char *) NULL)) < 0)
                logerr("Error on ioctl to detach from terminal");
            (void) close(fd);
        }
    }
}

```

```

        for (fd = 0; fd < n; fd++)
        {
            (void) close(fd);
        }
    }

/*
 * Compute size of STI CNB disk write buffer
 */
scnb_wrt_len = sizeof(shead) + (A1CNBLEN + (4 * A25CNBLEN));

/*
 * Create indexes for copies of each aper. STI CNBs to combined buffer
 */
scnb_cpy_idxs[0] = sizeof(shead);
scnb_cpy_idxs[1] = scnb_cpy_idxs[0] + A1CNBLEN;
scnb_cpy_idxs[2] = scnb_cpy_idxs[1] + A25CNBLEN;
scnb_cpy_idxs[3] = scnb_cpy_idxs[2] + A25CNBLEN;
scnb_cpy_idxs[4] = scnb_cpy_idxs[3] + A25CNBLEN;

/*
 * Fork process to write STI CNB data to disk
 */
if (!no_stis2dsk)
{
    if ((scnb_pid = fork()) == -1)    /* Fork error ? */
    {
        logerr("fork error");
        exit(-1);
    }
    else if (scnb_pid == 0)    /* Child */
    {
        (void) wrt_scnb();
    }
    num_children++;
}

/*
 * Fork process to send STI OTO data to drp
 */
if (!no_otos2drp)
{
    if ((soto_pid = fork()) == -1)    /* Fork error ? */
    {
        logerr("fork error");
        exit(-1);
    }
    else if (soto_pid == 0)    /* Child */
    {
        (void) snd_sotos();
    }
    num_children++;
}

/*
 * Fork process to send BIP OTO data to drp
 */
if (!no_bips2drp)
{
    if ((bip_pid = fork()) == -1)    /* Fork error ? */
    {
        logerr("fork error");
        exit(-1);
    }

```

```

    }
    else if (bip_pid == 0)      /* Child */
    {
        (void) snd_bips();
    }
    num_children++;
}

/*
 * Create shared memory
 */
(void) cshm();
fifo_level->scnbs_lost = 0;      /* Zero out lost data to drp/disk */
fifo_level->sotos_lost = 0;      /* (RC 11/22/94) */
fifo_level->bips_lost = 0;

/*
 * Zero FIFOs
 */
(void) bzero(scnb, (scnb_wrt_len * SCNB_DSK_FIFO_LEN));
(void) bzero(sotos, (sizeof(sti_oto_rec_t) * SOTO_DRP_FIFO_LEN));
(void) bzero(bips, (sizeof(bip_oto_rec_t) * BIP_DRP_FIFO_LEN));

/*
 * Set sti_oto_rec and bip_rec to start of output FIFOs
 */
sti_oto_rec = (sti_oto_rec_t *) sotos;
bip_rec = (bip_oto_rec_t *) bips;

/*
 * Put items that don't change during a run into arrays
 */
rd_array();                    /* Load array id in structures */
scnb_fifo_idx = 0;             /* Start at the start */

/*
 * Find the first space frame active aperture
 */
sf_1st_act_aper = -1;
for (i = 0; i < 2; i++)
{
    if (ap_bits[i] & exp_rcv_flags)
    {
        sf_1st_act_aper = i;
        break;
    }
}

/*
 * Find the first baffled cylinder active aperture
 */
bc_1st_act_aper = -1;
for (i = 2; i < 5; i++)
{
    if (ap_bits[i] & exp_rcv_flags)
    {
        bc_1st_act_aper = i;
        break;
    }
}

/*
 * Make sure there is an active aperture
 */
if((sf_1st_act_aper == -1) && (bc_1st_act_aper == -1))

```

```
{
    logmsg("startup: No active apertures specified");
    end_process();
}

return;
}
/*****/
```

Appendix K Listing of rd_array.c File

```

/* "@(#)rd_array.c 1.5" */
/*****
*
* ORNL I&C Division
* Module name: rd_array.c
* Version: 1.5
* Modified: 5/22/95 10:25:27
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
* Oak Ridge National Laboratory
* Instrumentation & Controls Division
* Real Time Computer Systems Group
* P O Box 2008, Bldg 3500, MS 6007
* Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FUNCTIONAL DESCRIPTION:
* Reads name of array
*
*
* STATUS RETURN VALUES
*
*
*****/
#include "rec_spls.h"
#define DEF_ARRAY_PATH "/oper/etc/ARRAY_ID"

void
rd_array()
{
    static char      SccsID[] = "@(#)rd_array.c 1.5 5/22/95 ORNL"; /* For ver
control */

    FILE            *fp;          /* File pointer */
    int              error; /* Error flag */
    int              i;           /* counter */
    char             path[64];
    char             *epathp;
    char             *sret;
    char             inbuf[16];    /* Buffer to read array_id to */

    error = 0;
    path[0] = 0;
    for (i = 0; i < 16; i++)
        inbuf[i] = 0;
    if ((epathp = getenv("ARRAY_PATH")) != NULL)
        sret = strcpy(path, epathp);
    else
        sret = strcpy(path, DEF_ARRAY_PATH);
    if ((fp = fopen(path, "r")) != NULL)
    {
        if (fscanf(fp, "%s", inbuf) < 0)
        {
            logerr("Error reading array id");
            error = 1;
        }
    }
    else
    {
        logerr("ARRAY_ID file open error");
        error = 1;
    }

    if (error)

```

```

{
    sret = strcpy(shead.array, "HAYES");
    for(i=0; i<SOTO_DRP_FIFO_LEN; i++)
        sret = strcpy(((sti_oto_rec_t *) (sotos + i*sizeof(sti_oto_rec_t)))
            ->head.array, "HAYES");
    for(i=0; i<BIP_DRP_FIFO_LEN; i++)
        sret = strcpy(((bip_oto_rec_t *) (bips + i*sizeof(bip_oto_rec_t)))
            ->head.array, "HAYES");
}
else
{
    sret = strcpy(shead.array, inbuf);

    for(i=0; i<SOTO_DRP_FIFO_LEN; i++)
        sret = strcpy(((sti_oto_rec_t *) (sotos + i*sizeof(sti_oto_rec_t)))
            ->head.array, inbuf);
    for(i=0; i<BIP_DRP_FIFO_LEN; i++)
        sret = strcpy(((bip_oto_rec_t *) (bips + i*sizeof(bip_oto_rec_t)))
            ->head.array, inbuf);
}
fclose(fp);
return;
}

/*****

```

Appendix L Listing of getargs.c File

```

/* "@(#)getargs.c 1.2" */
/*****
*
* ORNL I&C Division
* Module name:  getargs.c
* Version: 1.2
* Modified: 12/5/94 16:18:50
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
* Oak Ridge National Laboratory
* Instrumentation & Controls Division
* Real Time Computer Systems Group
* P O Box 2008, Bldg 3500, MS 6007
* Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FUNCTIONAL DESCRIPTION:
* Processes command line arguments for rec_spls
*
* STATUS RETURN VALUES
*
*****/
#include "rec_spls.h"

void
getargs(argc, argv)
    int      argc;
    char     *argv[];
{
    static char  SccsID[] = "@(#)getargs.c 1.2 12/5/94 ORNL"; /* For ver
control */

    extern char  *optarg; /* Used by getopt stuff */
    extern int   optind; /* Used by getopt stuff */
    extern int   opterr; /* Used by getopt stuff */
    int          c;
    int          errflg = 0; /* Error flag */

    while ((c = getopt(argc, argv, "dsoba:m:")) != -1)
        switch (c)
        {
            case 'd':
                detach = 1;
                break;
            case 's':
                no_stis2dsk = 1;
                break;
            case 'o':
                no_otos2drp = 1;
                break;
            case 'b':
                no_bips2drp = 1;
                break;
            case 'a':
                exp_rcv_flags = (unsigned int) atoi(optarg);
                break;
            case 'm':
                buf_ms = (unsigned int) atoi(optarg);
                break;
            case '?':
                errflg++;
        }
}

```

```

if (errflg)
{
(void) logmsg("\nUsage  rec_spls [-d] [-s] [-t] [-b] [-a mask]\n");
(void) logmsg("          -d  detach process\n");
(void) logmsg("          -s  inhibit STI writes to disk\n");
(void) logmsg("          -o  inhibit OTO writes to drp\n");
(void) logmsg("          -b  inhibit BIP writes to drp\n");
(void) logmsg("          -a  specify active aperture mask\n");
(void) logmsg("          -m  specify buffer milestone interval\n\n");
    exit(-1);
}

return;
}
/*****/

```

Appendix M Listing of cshm.c File

```

/* "(@#)cshm.c      1.3" */
/*****
*
*                                     ORNL I&C Division
* Module name:  cshm.c
* Version: 1.3
* Modified: 1/13/95 15:35:12
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
*             Oak Ridge National Laboratory
*             Instrumentation & Controls Division
*             Real Time Computer Systems Group
*             P O Box 2008, Bldg 3500, MS 6007
*             Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FUNCTIONAL DESCRIPTION:
*     Creates and attaches to shared memory
*
*
* STATUS RETURN VALUES
*
*
*****/
#include "rec_spls.h"

void
cshm()
{
    static char      SccsID[] = "(@#)cshm.c      1.3 1/13/95 ORNL"; /* For ver
control */

    ctlsmid = shmget((key_t) CTL_SHMKEY, sizeof(fifo_level_t),
                     SHM_PERMS | IPC_CREAT);
    if (ctlsmid < 0)
    {
        logerr("ctl sm create failed");
        exit(-1);
    }

    if ((fifo_level = (fifo_level_t *) shmat(ctlsmid, (char *) 0,
                                              0)) == (fifo_level_t *) - 1)
    {
        logerr("fifo level sm attach failed");
        exit(-1);
    }

    scnbsmid = shmget((key_t) SCNB_SHMKEY, (scnb_wrt_len * SCNB_DSK_FIFO_LEN),
                     SHM_PERMS | IPC_CREAT);
    if (scnbsmid < 0)
    {
        logerr("scnb sm create failed");
        exit(-1);
    }

    if ((scnb = (char *) shmat(scnbsmid, (char *) 0, 0)) == (char *) -1)
    {
        logerr("scnb sm attach failed");
        exit(-1);
    }

    sotosmid = shmget((key_t) SOTO_SHMKEY,
                     (sizeof(sti_oto_rec_t) * SOTO_DRP_FIFO_LEN),
                     SHM_PERMS | IPC_CREAT);

```

```

    if (sotosmid < 0)
    {
        logerr("soto sm create failed");
        exit(-1);
    }
    if ((sotos = (char *) shmat(sotosmid, (char *) 0, 0)) == (char *) -1)
    {
        logerr("soto sm attach failed");
        exit(-1);
    }

    bipsmid = shmget((key_t) BIP_SHMKEY, (sizeof(bip_oto_rec_t) *
BIP_DRP_FIFO_LEN),
        SHM_PERMS | IPC_CREAT);
    if (bipsmid < 0)
    {
        logerr("bip sm create failed");
        exit(-1);
    }
    if ((bips = (char *) shmat(bipsmid, (char *) 0, 0)) == (char *) -1)
    {
        logerr("bip sm attach failed");
        exit(-1);
    }

    frame_smid = shmget((key_t) FRAME_SHMKEY, sizeof(int), SHM_PERMS |
IPC_CREAT);
    if (frame_smid < 0)
    {
        logerr("frame sm create failed");
        exit(-1);
    }
    if ((frame = (int *) shmat(frame_smid, (char *) 0, 0)) == (int *) -1)
    {
        logerr("frame sm attach failed");
        exit(-1);
    }

    stime_smid = shmget((key_t) STIME_SHMKEY, sizeof(stime_t),
        SHM_PERMS | IPC_CREAT);
    if (stime_smid < 0)
    {
        logerr("stime sm create failed");
        exit(-1);
    }
    if ((stime = (stime_t *) shmat(stime_smid, (char *) 0, 0)) == (stime_t *) -
1)
    {
        logerr("stime sm attach failed");
        exit(-1);
    }

    store_smid = shmget((key_t) STORE_SHMKEY, sizeof(int), SHM_PERMS |
IPC_CREAT);
    if (store_smid < 0)
    {
        logerr("store sm create failed");
        exit(-1);
    }
    if ((store = (int *) shmat(store_smid, (char *) 0, 0)) == (int *) -1)
    {
        logerr("store sm attach failed");
        exit(-1);
    }

```

```
    return;  
}  
/*****/
```

Appendix N Listing of open_hippi.c File

```

/* "@(#)open_hippi.c      1.2" */
/*****
*
*                                     ORNL I&C Division
* Module name:  open_hippi.c
* Version: 1.2
* Modified: 12/5/94 16:22:52
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
*             Oak Ridge National Laboratory
*             Instrumentation & Controls Division
*             Real Time Computer Systems Group
*             P O Box 2008, Bldg 3500, MS 6007
*             Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FUNCTIONAL DESCRIPTION:
*   Opens Hippi Channel
*
*
* STATUS RETURN VALUES
*   None, Hippi file descriptor is global
*
*****/
#include "rec_spls.h"
#define IFIELD 4

void open_hippi();

void
open_hippi()
{
    static char    SccsID[] = "@(#)open_hippi.c      1.2 12/5/94 ORNL"; /* For
ver control */

    void            end_process();
    int             one31 = 131;          /* Used to set HIPPI ULPID */

    /*
    * Open Hippi for read from APPs
    */
    if ((hfd = open(HIP_PATH, O_RDWR | O_EXCL)) < 0)
    {
        logmsg("Opening hippi, no reset");
        if ((hfd = open(HIP_PATH, O_RDONLY)) < 0)
        {
            logerr("Hippi open error");
            end_process();
        }
    }
    else
        reset_hippi();

    (void) hip_set_parms(hfd, 0, 0, 0, 0, IFIELD, 0, 0);
    if (ioctl(hfd, HIP_ATTACH, &one31) < 0)
    {
        logerr("ioctl[HIP_attach]");
        end_process;
    }

    return;
}
/*****

```

Appendix O Listing of reset_hippi.c File

```

/* "@(#)reset_hippi.c      1.2" */
/*****
*
* ORNL I&C Division
* Module name:  reset_hippi.c
* Version: 1.2
* Modified: 12/5/94 16:25:13
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
*             Oak Ridge National Laboratory
*             Instrumentation & Controls Division
*             Real Time Computer Systems Group
*             P O Box 2008, Bldg 3500, MS 6007
*             Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FUNCTIONAL DESCRIPTION:
*   Resets Hippi device
*
* STATUS RETURN VALUES
*
*****/
#include "rec_spls.h"

void
reset_hippi()
{
    static char    SccsID[] = "@(#)reset_hippi.c      1.2 12/5/94 ORNL"; /* For
ver control */

    logmsg("Resetting hippi");
    if (ioctl(hfd, HIP_TXHANG, 0) < 0)
        logerr("Hippi Tx reset");
    if (ioctl(hfd, HIP_RXHANG, 0) < 0)
        logerr("Hippi Rx reset");
    (void) close(hfd);

    hfd = open(HIP_PATH, O_RDONLY);
    if (hfd == -1)
    {
        logerr("Hippi open (reset_hippi) error");
        exit(1);
    }
    sleep(1);
    return;
}

/*****/

```

Appendix P Listing of hip_set_params.c File

```

/* "@(#)hip_set_parms.c    1.2" */
/*****
*
*                                     ORNL I&C Division
* Module name:  hip_set_parms.c
* Version: 1.2
* Modified: 12/5/94 16:20:57
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
*             Oak Ridge National Laboratory
*             Instrumentation & Controls Division
*             Real Time Computer Systems Group
*             P O Box 2008, Bldg 3500, MS 6007
*             Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FUNCTION
*   hip_set_parms - Sets Hippi device operational parameters.
*
*   This is used to set the operational parameters of the driver
*   through the ioctl function. Any non-negative parameters
*   are passed to the driver. Created from the Chi HipSetParms
*   function.
*
* STATUS RETURN VALUES
*   0 success
*  -1 error
*
*****/
#include "rec_spls.h"

int
hip_set_parms(pd, mode, tto, dlz, txifv, rxifv, txulpid, rxulpid)

int          pd;          /* File descriptor */
int          mode;        /* mode */
int          tto;         /* transfer timeout */
int          dlz;         /* D1 size */
int          txifv;       /* Transmit I-Field */
int          rxifv;       /* Receive I-Field */
int          txulpid;     /* Transmit Upper layer protocol ID */
int          rxulpid;     /* Receive Upper layer protocol ID */
{
    static char  SccsID[] = "@(#)hip_set_parms.c    1.2 12/5/94 ORNL"; /* For
ver control */

    struct hipio  hipio;

    if (ioctl(pd, HIP_GET, &hipio) < 0)
    {
        logerr("ioctl[HIP_GET]");
        return 1;
    }

    if (mode >= 0)
        hipio.mode = (u_int) mode;
    if (dlz >= 0)
        hipio.dlz = dlz;
    if (tto >= 0)
        hipio.tto = tto;
    if (txulpid >= 0)
        hipio.tx_ulpid = txulpid;
    if (rxulpid >= 0)

```

```

        hipio.rx_ulpid = rxulpid;
    if (txifv != -1)
        hipio.tx_ifv = txifv;
    if (rxifv != -1)
        hipio.rx_ifv = rxifv;

    if (ioctl(pd, HIP_SET, &hipio) < 0)
    {
        logerr("ioctl[HIP_SET]");
        return -1;
    }
    return 0;
}

/*****

```

Appendix Q Listing of process_pkt.c File

```

/* "(@"#)process_pkt.c      1.9" */
/*****
*
*                                     ORNL I&C Division
* Module name:  process_pkt.c
* Version: 1.9
* Modified: 10/4/95 12:39:04
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
*             Oak Ridge National Laboratory
*             Instrumentation & Controls Division
*             Real Time Computer Systems Group
*             P O Box 2008, Bldg 3500, MS 6007
*             Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FUNCTIONAL DESCRIPTION:
*   This function process one packet from APP0
*
* STATUS RETURN VALUES
*
*****/
#include "rec_spls.h"

#define A1BIT 1
#define A2BIT 2
#define A3BIT 4
#define A4BIT 8
#define A5BIT 16

void
process_pkt(cbuf)
int      cbuf; /* Input buffer being processed */
{
    static char      SccsID[] = "(@"#)process_pkt.c      1.9 10/4/95 ORNL"; /* For
ver control */

    static          ap_bits[] = {A1BIT, A2BIT, A3BIT, A4BIT, A5BIT};
    int             ap_idx; /* Aperture index passed to merge function */
    static int      before = 0; /* (TFG mod) Frame counts start at 2 */
    void            end_process();
    int             i, j, k; /* Indexes */
    void            merge_ap(); /* Declare function */
    void            process_pkt(); /* Declare function */
    char            *retp; /* Return pointer for memcpy */
    static int      set_sync_time = 1; /* Flag reset when sync time read */
    int             tmp_buff_num; /* Holds buffer number */
    int             aper_buff_num[5]; /* Holds buffer number for each aperture */
    static int      time_set_cntr=0; /* counts the settings of time */

    if ((set_sync_time) && ((stime->track_pid) > 0))
    {
        /*
        * Copy sync time to where used
        */
        shead.time.year = sti_oto_rec->head.time.year =
            bip_rec->head.time.year = stime->bft.year;
        shead.time.day = sti_oto_rec->head.time.day =
            bip_rec->head.time.day = stime->bft.day;
    }
}

```

```

shead.time.hour = sti_oto_rec->head.time.hour =
    bip_rec->head.time.hour = stime->bft.hour;
shead.time.minute = sti_oto_rec->head.time.minute =
    bip_rec->head.time.minute = stime->bft.minute;
shead.time.sec = sti_oto_rec->head.time.sec =
    bip_rec->head.time.sec = stime->bft.sec;
shead.time.millisec = sti_oto_rec->head.time.millisec =
    bip_rec->head.time.millisec = stime->bft.millisec;

/* increment time set counter */
time_set_cntr++;

/* If the time has been written into all SOTO and BIP headers then
   don't write time data into header again */
if(time_set_cntr >= SOTO_DRP_FIFO_LEN &&
    time_set_cntr >= BIP_DRP_FIFO_LEN)
    set_sync_time = 0;
}

tmp_buff_num = -1;

/*
 * Process space frame steering data and buffer number
 */
if (sf_1st_act_aper != -1)
{
    k = sf_1st_act_aper;

    if(k == 0) {
        /* aperture 1 is first active spaceframe aperture */
        tmp_buff_num = inbuf[cbuf].al.head.buf_num;

        shead.sf_range = inbuf[cbuf].al.steer.range;
        shead.sf_azimuth = inbuf[cbuf].al.steer.azimuth;
        shead.sf_dep_elev = inbuf[cbuf].al.steer.dep_elev;
        shead.sf_measr_azim = inbuf[cbuf].al.beam_steer.measr_azim;
        shead.sf_measr_elev = inbuf[cbuf].al.beam_steer.measr_elev;
        shead.sf_image_azim = inbuf[cbuf].al.beam_steer.image_azim;
        shead.sf_image_elev = inbuf[cbuf].al.beam_steer.image_elev;
        for (i = 0; i < 5 ; i++)
            shead.sf_measr_hoff[i] = inbuf[cbuf].al.beam_steer.measr_hoff[i];
        for (i = 0; i < 2 ; i++)
            shead.sf_measr_voff[i] = inbuf[cbuf].al.beam_steer.measr_voff[i];

        sti_oto_rec->head.sf_range = inbuf[cbuf].al.steer.range;
        sti_oto_rec->head.sf_azimuth = inbuf[cbuf].al.steer.azimuth;
        sti_oto_rec->head.sf_dep_elev = inbuf[cbuf].al.steer.dep_elev;
        sti_oto_rec->head.sf_measr_azim = inbuf[cbuf].al.beam_steer.measr_azim;
        sti_oto_rec->head.sf_measr_elev = inbuf[cbuf].al.beam_steer.measr_elev;
        sti_oto_rec->head.sf_image_azim = inbuf[cbuf].al.beam_steer.image_azim;
        sti_oto_rec->head.sf_image_elev = inbuf[cbuf].al.beam_steer.image_elev;
        for (i = 0; i < 5 ; i++)
            sti_oto_rec->head.sf_measr_hoff[i] =
                inbuf[cbuf].al.beam_steer.measr_hoff[i];
        for (i = 0; i < 2 ; i++)
            sti_oto_rec->head.sf_measr_voff[i] =
                inbuf[cbuf].al.beam_steer.measr_voff[i];

        bip_rec->head.sf_range = inbuf[cbuf].al.steer.range;
        bip_rec->head.sf_azimuth = inbuf[cbuf].al.steer.azimuth;
        bip_rec->head.sf_dep_elev = inbuf[cbuf].al.steer.dep_elev;
        bip_rec->head.sf_measr_azim = inbuf[cbuf].al.beam_steer.measr_azim;
        bip_rec->head.sf_measr_elev = inbuf[cbuf].al.beam_steer.measr_elev;
        bip_rec->head.sf_image_azim = inbuf[cbuf].al.beam_steer.image_azim;
        bip_rec->head.sf_image_elev = inbuf[cbuf].al.beam_steer.image_elev;
    }
}

```

```

    for (i = 0; i < 5 ; i++)
        bip_rec->head.sf_measr_hoff[i] =
            inbuf[cbuf].a1.beam_steer.measr_hoff[i];
    for (i = 0; i < 2 ; i++)
        bip_rec->head.sf_measr_voff[i] =
            inbuf[cbuf].a1.beam_steer.measr_voff[i];
}
else{
    /* aperture 2 is first active spaceframe aperture */
    tmp_buff_num = inbuf[cbuf].a2_5[0].head.buf_num;

    shead.sf_range = inbuf[cbuf].a2_5[0].steer.range;
    shead.sf_azimuth = inbuf[cbuf].a2_5[0].steer.azimuth;
    shead.sf_dep_elev = inbuf[cbuf].a2_5[0].steer.dep_elev;
    shead.sf_measr_azim = inbuf[cbuf].a2_5[0].beam_steer.measr_azim;
    shead.sf_measr_elev = inbuf[cbuf].a2_5[0].beam_steer.measr_elev;
    shead.sf_image_azim = inbuf[cbuf].a2_5[0].beam_steer.image_azim;
    shead.sf_image_elev = inbuf[cbuf].a2_5[0].beam_steer.image_elev;
    for (i = 0; i < 5 ; i++)
        shead.sf_measr_hoff[i]=inbuf[cbuf].a2_5[0].beam_steer.measr_hoff[i];
    for (i = 0; i < 2 ; i++)
        shead.sf_measr_voff[i]=inbuf[cbuf].a2_5[0].beam_steer.measr_voff[i];

    sti_oto_rec->head.sf_range = inbuf[cbuf].a2_5[0].steer.range;
    sti_oto_rec->head.sf_azimuth = inbuf[cbuf].a2_5[0].steer.azimuth;
    sti_oto_rec->head.sf_dep_elev = inbuf[cbuf].a2_5[0].steer.dep_elev;
    sti_oto_rec->head.sf_measr_azim =
        inbuf[cbuf].a2_5[0].beam_steer.measr_azim;
    sti_oto_rec->head.sf_measr_elev =
        inbuf[cbuf].a2_5[0].beam_steer.measr_elev;
    sti_oto_rec->head.sf_image_azim =
        inbuf[cbuf].a2_5[0].beam_steer.image_azim;
    sti_oto_rec->head.sf_image_elev =
        inbuf[cbuf].a2_5[0].beam_steer.image_elev;
    for (i = 0; i < 5 ; i++)
        sti_oto_rec->head.sf_measr_hoff[i] =
            inbuf[cbuf].a2_5[0].beam_steer.measr_hoff[i];
    for (i = 0; i < 2 ; i++)
        sti_oto_rec->head.sf_measr_voff[i] =
            inbuf[cbuf].a2_5[0].beam_steer.measr_voff[i];

    bip_rec->head.sf_range = inbuf[cbuf].a2_5[0].steer.range;
    bip_rec->head.sf_azimuth = inbuf[cbuf].a2_5[0].steer.azimuth;
    bip_rec->head.sf_dep_elev = inbuf[cbuf].a2_5[0].steer.dep_elev;
    bip_rec->head.sf_measr_azim = inbuf[cbuf].a2_5[0].beam_steer.measr_azim;
    bip_rec->head.sf_measr_elev = inbuf[cbuf].a2_5[0].beam_steer.measr_elev;
    bip_rec->head.sf_image_azim = inbuf[cbuf].a2_5[0].beam_steer.image_azim;
    bip_rec->head.sf_image_elev = inbuf[cbuf].a2_5[0].beam_steer.image_elev;
    for (i = 0; i < 5 ; i++)
        bip_rec->head.sf_measr_hoff[i] =
            inbuf[cbuf].a2_5[0].beam_steer.measr_hoff[i];
    for (i = 0; i < 2 ; i++)
        bip_rec->head.sf_measr_voff[i] =
            inbuf[cbuf].a2_5[0].beam_steer.measr_voff[i];
}
}
else
{
    shead.sf_range = 0.0;
    shead.sf_azimuth = 0.0;
    shead.sf_dep_elev = 0.0;
    shead.sf_measr_azim = 0.0;
    shead.sf_measr_elev = 0.0;
    shead.sf_image_azim = 0.0;
    shead.sf_image_elev = 0.0;

```

```

for (i = 0; i < 5 ; i++)
    shead.sf_measr_hoff[i] = 0.0;
for (i = 0; i < 2 ; i++)
    shead.sf_measr_voff[i] = 0.0;

sti_oto_rec->head.sf_range = 0.0;
sti_oto_rec->head.sf_azimuth = 0.0;
sti_oto_rec->head.sf_dep_elev = 0.0;
sti_oto_rec->head.sf_measr_azim = 0.0;
sti_oto_rec->head.sf_measr_elev = 0.0;
sti_oto_rec->head.sf_image_azim = 0.0;
sti_oto_rec->head.sf_image_elev = 0.0;
for (i = 0; i < 5 ; i++)
    sti_oto_rec->head.sf_measr_hoff[i] = 0.0;
for (i = 0; i < 2 ; i++)
    sti_oto_rec->head.sf_measr_voff[i] = 0.0;

bip_rec->head.sf_range = 0.0;
bip_rec->head.sf_azimuth = 0.0;
bip_rec->head.sf_dep_elev = 0.0;
bip_rec->head.sf_measr_azim = 0.0;
bip_rec->head.sf_measr_elev = 0.0;
bip_rec->head.sf_image_azim = 0.0;
bip_rec->head.sf_image_elev = 0.0;
for (i = 0; i < 5 ; i++)
    bip_rec->head.sf_measr_hoff[i] = 0.0;
for (i = 0; i < 2 ; i++)
    bip_rec->head.sf_measr_voff[i] = 0.0;
}

/*
 * Process baffled cylinder steering data and buffer number
 */
if (bc_1st_act_aper != -1)
{
    k = bc_1st_act_aper;
    tmp_buff_num = inbuf[cbuf].a2_5[k-1].head.buf_num;

    shead.bc_range = inbuf[cbuf].a2_5[k-1].steer.range;
    shead.bc_azimuth = inbuf[cbuf].a2_5[k-1].steer.azimuth;
    shead.bc_dep_elev = inbuf[cbuf].a2_5[k-1].steer.dep_elev;
    shead.bc_measr_azim = inbuf[cbuf].a2_5[k-1].beam_steer.measr_azim;
    shead.bc_measr_elev = inbuf[cbuf].a2_5[k-1].beam_steer.measr_elev;
    shead.bc_image_azim = inbuf[cbuf].a2_5[k-1].beam_steer.image_azim;
    shead.bc_image_elev = inbuf[cbuf].a2_5[k-1].beam_steer.image_elev;
    for (i = 0; i < 5 ; i++)
        shead.bc_measr_hoff[i] =
            inbuf[cbuf].a2_5[k-1].beam_steer.measr_hoff[i];
    for (i = 0; i < 2 ; i++)
        shead.bc_measr_voff[i] =
            inbuf[cbuf].a2_5[k-1].beam_steer.measr_voff[i];

    sti_oto_rec->head.bc_range = inbuf[cbuf].a2_5[k-1].steer.range;
    sti_oto_rec->head.bc_azimuth = inbuf[cbuf].a2_5[k-1].steer.azimuth;
    sti_oto_rec->head.bc_dep_elev = inbuf[cbuf].a2_5[k-1].steer.dep_elev;
    sti_oto_rec->head.bc_measr_azim =
        inbuf[cbuf].a2_5[k-1].beam_steer.measr_azim;
    sti_oto_rec->head.bc_measr_elev =
        inbuf[cbuf].a2_5[k-1].beam_steer.measr_elev;
    sti_oto_rec->head.bc_image_azim =
        inbuf[cbuf].a2_5[k-1].beam_steer.image_azim;
    sti_oto_rec->head.bc_image_elev =
        inbuf[cbuf].a2_5[k-1].beam_steer.image_elev;
    for (i = 0; i < 5 ; i++)
        sti_oto_rec->head.bc_measr_hoff[i] =

```

```

        inbuf[cbuf].a2_5[k-1].beam_steer.mesr_hoff[i];
    for (i = 0; i < 2 ; i++)
        sti_oto_rec->head.bc_mesr_voff[i] =
            inbuf[cbuf].a2_5[k-1].beam_steer.mesr_voff[i];

    bip_rec->head.bc_range = inbuf[cbuf].a2_5[k-1].steer.range;
    bip_rec->head.bc_azimuth = inbuf[cbuf].a2_5[k-1].steer.azimuth;
    bip_rec->head.bc_dep_elev = inbuf[cbuf].a2_5[k-1].steer.dep_elev;
    bip_rec->head.bc_mesr_azim = inbuf[cbuf].a2_5[k-1].beam_steer.mesr_azim;
    bip_rec->head.bc_mesr_elev = inbuf[cbuf].a2_5[k-1].beam_steer.mesr_elev;
    bip_rec->head.bc_image_azim = inbuf[cbuf].a2_5[k-1].beam_steer.image_azim;
    bip_rec->head.bc_image_elev = inbuf[cbuf].a2_5[k-1].beam_steer.image_elev;
    for (i = 0; i < 5 ; i++)
        bip_rec->head.bc_mesr_hoff[i] =
            inbuf[cbuf].a2_5[k-1].beam_steer.mesr_hoff[i];
    for (i = 0; i < 2 ; i++)
        bip_rec->head.bc_mesr_voff[i] =
            inbuf[cbuf].a2_5[k-1].beam_steer.mesr_voff[i];
}
else
{
    shead.bc_range = 0.0;
    shead.bc_azimuth = 0.0;
    shead.bc_dep_elev = 0.0;
    shead.bc_mesr_azim = 0.0;
    shead.bc_mesr_elev = 0.0;
    shead.bc_image_azim = 0.0;
    shead.bc_image_elev = 0.0;
    for (i = 0; i < 5 ; i++)
        shead.bc_mesr_hoff[i] = 0.0;
    for (i = 0; i < 2 ; i++)
        shead.bc_mesr_voff[i] = 0.0;

    sti_oto_rec->head.bc_range = 0.0;
    sti_oto_rec->head.bc_azimuth = 0.0;
    sti_oto_rec->head.bc_dep_elev = 0.0;
    sti_oto_rec->head.bc_mesr_azim = 0.0;
    sti_oto_rec->head.bc_mesr_elev = 0.0;
    sti_oto_rec->head.bc_image_azim = 0.0;
    sti_oto_rec->head.bc_image_elev = 0.0;
    for (i = 0; i < 5 ; i++)
        sti_oto_rec->head.bc_mesr_hoff[i] = 0.0;
    for (i = 0; i < 2 ; i++)
        sti_oto_rec->head.bc_mesr_voff[i] = 0.0;

    bip_rec->head.bc_range = 0.0;
    bip_rec->head.bc_azimuth = 0.0;
    bip_rec->head.bc_dep_elev = 0.0;
    bip_rec->head.bc_mesr_azim = 0.0;
    bip_rec->head.bc_mesr_elev = 0.0;
    bip_rec->head.bc_image_azim = 0.0;
    bip_rec->head.bc_image_elev = 0.0;
    for (i = 0; i < 5 ; i++)
        bip_rec->head.bc_mesr_hoff[i] = 0.0;
    for (i = 0; i < 2 ; i++)
        bip_rec->head.bc_mesr_voff[i] = 0.0;
}

shead.buf_num = tmp_buff_num;
sti_oto_rec->head.buf_num = tmp_buff_num;
bip_rec->head.buf_num = tmp_buff_num;

/*
 * Put header in CNB STI bufer for write
 */

```

```

retp = memcpy(scnb + scnb_wrt_len * scnb_fifo_idx,
              (char *) &shead, sizeof(shead));

/*
 * Put buffer number in shared memory and signal processes in group so
 * they know it has been updated
 */
*frame = tmp_buff_num;

/*
 * Check to see that buffer numbers are increasing one at a time
 */
if (++before != *frame)
    logmsg("frame number sequence error between %i and %i",
          before, *frame);
before = *frame;

/*
 * Signal tracking proces if it exist
 */
if (stime->track_pid > 2)
{
    if (kill(stime->track_pid, SIGUSR1) < 0)
    {
        logerr("Error sending buffer update signal");
    }
}

/*
 * Clear custom BIP summing area
 */
{
    for (i = 0; i < NBIPS; i++)
    {
        for (k = NOTOS; k < (NOTOS + NCOTOS); k++)
        {
            bip_rec->bip_otos[i][k] = 0.0;
        }
    }
}

/*
 * Merge data from all active apertures, and check that all apertures in
 * this packet have the same buffer number
 */
i = 0; /* Used to flag an aperture number error */
for (ap_idx = 0; ap_idx < 5; ap_idx++)
{
    if (ap_bits[ap_idx] & exp_rcv_flags)
    {
        merge_ap(ap_idx, cbuf);

        if (ap_idx == 0)
            aper_buff_num[ap_idx] = inbuf[cbuf].a1.head.buf_num;
        else
            aper_buff_num[ap_idx] = inbuf[cbuf].a2_5[ap_idx-1].head.buf_num;

        if (aper_buff_num[ap_idx] != *frame)
        {
            i = 1;
        }
    }
}

/*

```

```

    * If all apertures did not contain same buffer number log them
    */
if (i)
{
    for (i = 0; i < 5; i++)
    {
        logmsg("Aperture %i buffer number = %i", (i + 1), aper_buff_num[i]);
    }
}

/*
 * Write data to DRP and Disk FIFOs
 */
wrt_fifos();
return;
}

/*****

```

Appendix R Listing of merge_ap.c File

```

/* "@(#)merge_ap.c 1.8" */
/*****
*
* ORNL I&C Division
* Module name: merge_ap.c
* Version: 1.8
* Modified: 11/20/95 14:47:36
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
* Oak Ridge National Laboratory
* Instrumentation & Controls Division
* Real Time Computer Systems Group
* P O Box 2008, Bldg 3500, MS 6007
* Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FUNCTIONAL DESCRIPTION:
* Merge data from one aperture to structures used for DRP
* and Disk writes
*
* STATUS RETURN VALUES
*
*
*****/
#include "rec_spls.h"
/*
* The following define which OTOs come from which aperture
*/
#define AP1_OTO_START 0
#define AP1_OTO_END 12
#define AP2_OTO_START 13
#define AP2_OTO_END 15
#define AP3_OTO_START 16
#define AP3_OTO_END 18
#define AP4_OTO_START 19
#define AP4_OTO_END 21
#define AP5_OTO_START 22
#define AP5_OTO_END 24

void
merge_ap(ap_idx, cbuf)
    int ap_idx; /* Aperture to process */
    int cbuf; /* Buffer in inbuf[] being worked */
{
    static char SccsID[] = "@(#)merge_ap.c 1.8"; /* For ver control */

    FILE *fout, *fopen();
    int file_write_siz;
    int file_stat;

    static aperture_oto_positions_t ap_oto_p[] =
    {
        AP1_OTO_START, AP1_OTO_END,
        AP2_OTO_START, AP2_OTO_END,
        AP3_OTO_START, AP3_OTO_END,
        AP4_OTO_START, AP4_OTO_END,
        AP5_OTO_START, AP5_OTO_END
    };
    int i, j, k; /* Indexes */
    char *retp; /* Pointer for return from copy */
    static int scnblen[] = /* Array of CNB lengths for easy access */
    {
        A1CNBLEN,

```

```

A25CNBLEN,
A25CNBLEN,
A25CNBLEN,
A25CNBLEN
};

if (ap_idx == 0)
{
    /*
     * Copy CNB beam data to temp buffer for disk write
     */
    retp = memcpy((scnb + (scnb_fifo_idx * scnb_wrt_len)
                  + scnb_cpy_idxs[(ap_idx)]),
                  (char *) inbuf[cbuf].a1.scnb, scnbllen[(ap_idx)]);

    /*
     * Put STI OTOS in combined array
     */
    for (i = 0; i < NUM_HBEAMS; i++) /* Copy STI OTOS */
    {
        for (j = 0; j < NUM_VBEAMS; j++)
        {
            for (k = ap_oto_p[ap_idx].start; k <=
                  ap_oto_p[ap_idx].end; k++)
            {
                sti_oto_rec->sti_otos[i][j][k] =
                    inbuf[cbuf].a1.sotos[i][j][k];
            }
        }
    }

    /*
     * Put BIP OTOS in combined array
     */
    for (i = 0; i < NBIPS; i++)
    {
        for (k = ap_oto_p[ap_idx].start; k <= ap_oto_p[ap_idx].end; k++)
        {
            bip_rec->bip_otos[i][k] = inbuf[cbuf].a1.botos[i][k];
        }
    }

    /*
     * Put custom BIP OTOS in combined array
     */
    for (i = 0; i < NBIPS; i++)
    {
        for (k = 0; k < NCOTOS; k++)
        {
            bip_rec->bip_otos[i][k + NOTOS] +=
                inbuf[cbuf].a1.bcotos[i][k];
        }
    }
} /* if(ap_idx == 0) */
else
{
    /*
     * Copy CNB beam data to temp buffer for disk write
     */
    retp = memcpy((scnb + (scnb_fifo_idx * scnb_wrt_len)
                  + scnb_cpy_idxs[(ap_idx)]),
                  (char *) inbuf[cbuf].a2_5[ap_idx - 1].scnb, scnbllen[(ap_idx)]);
}

```

```

/*
 * Put STI OTOS in combined array
 */
for (i = 0; i < NUM_HBEAMS; i++) /* Copy STI OTOS */
{
    for (j = 0; j < NUM_VBEAMS; j++)
    {
        for (k = ap_oto_p[ap_idx].start; k <=
            ap_oto_p[ap_idx].end; k++)
        {
            sti_oto_rec->sti_otos[i][j][k] =
                inbuf[cbuf].a2_5[ap_idx - 1].sotos[i][j][k -
                    ap_oto_p[ap_idx].start];
        }
    }
}

/*
 * Put BIP OTOS in combined array
 */
for (i = 0; i < NBIPS; i++)
{
    for (k = ap_oto_p[ap_idx].start; k <= ap_oto_p[ap_idx].end; k++)
    {
        bip_rec->bip_otos[i][k] =
            inbuf[cbuf].a2_5[ap_idx - 1].botos[i][k -
                ap_oto_p[ap_idx].start];
    }
}

/*
 * Put custom BIP OTOS in combined array
 */
for (i = 0; i < NBIPS; i++)
{
    for (k = 0; k < NCOTOS; k++)
    {
        bip_rec->bip_otos[i][k + NOTOS] +=
            inbuf[cbuf].a2_5[ap_idx - 1].bcotos[i][k];
    }
}

return;
}

/*****

```

Appendix S Listing of wrt_fifos.c File

```

/* "@(#)wrt_fifos.c 1.4" */
/*****
*
* ORNL I&C Division
* Module name: wrt_fifos.c
* Version: 1.4
* Modified: 1/13/95 15:19:03
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
* Oak Ridge National Laboratory
* Instrumentation & Controls Division
* Real Time Computer Systems Group
* P O Box 2008, Bldg 3500, MS 6007
* Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FUNCTIONAL DESCRIPTION:
* Write data to DRP and Disk FIFOs
*
* This function updates the FIFOs used to sent data out on DRP
* and to disk. Flags set by command line arguments are used
* to enable or disable the writes. If the FIFO fills up, a
* message is logged and writes are suspended until room is
* available in the FIFO, at which point another message is logged.
*
* STATUS RETURN VALUES
* NONE
*
*****/
#include "rec_spls.h"

int
wrt_fifos()
{
    static char SccsID[] = "@(#)wrt_fifos.c 1.4 1/13/95 ORNL"; /* For ver
control */

    char *mr; /* For memcpy return */
    static int bip_fifo_full = 0; /* Flag that bip FIFO is full */
    static int scnb_fifo_full = 0; /* Flag that scnb FIFO is full */
    static int soto_fifo_full = 0; /* Flag that soto FIFO is full */

    sti_oto_rec->head.apex = exp_rcv_flags;
    sti_oto_rec->head.apval = exp_rcv_flags;

    /*
    * Update shared memory to tell child process new buffer is ready for disk
    * write
    */
    if(!no_stis2dsk)
    {
        if(fifo_level->scnb == SCNB_DSK_FIFO_LEN)
        {
            (fifo_level->scnbs_lost)++; /* Added for monitoring (RC 11/17/94) */
            if(!scnb_fifo_full)
            {
                scnb_fifo_full = 1;
                logmsg("wrt_fifos: STI CNB FIFO full, data not being written");
            }
        }
        else
        {
            if(scnb_fifo_full)
            {

```

```

        scnb_fifo_full = 0;
        logmsg("wrt_fifos: STI CNB FIFO data being written again");
    }
    scnb_fifo_idx = (++scnb_fifo_idx) % SCNB_DSK_FIFO_LEN;
    (fifo_level->scnb)++;
}

/*
 * Update sti_oto_rec pointer to new FIFO buffer and manage indexes.
 */
if(!no_otos2drp)
{
    if(fifo_level->soto == SOTO_DRP_FIFO_LEN)
    {
        (fifo_level->sotos_lost)++; /* Added for monitoring (RC 11/17/94) */
        if(!soto_fifo_full)
        {
            soto_fifo_full = 1;
            logmsg("wrt_fifos: STI OTO FIFO full, data not being written");
        }
    }
    else
    {
        if(soto_fifo_full)
        {
            soto_fifo_full = 0;
            logmsg("wrt_fifos: STI OTO FIFO data being written again");
        }
        soto_fifo_idx = (++soto_fifo_idx) % SOTO_DRP_FIFO_LEN;
        sti_oto_rec = (sti_oto_rec_t *) (sotos + (sizeof(sti_oto_rec_t) *
            soto_fifo_idx));
        (fifo_level->soto)++;
    }
}

/*
 * Update bip_rec pointer and manage indexes.
 */
if(!no_bips2drp)
{
    if(fifo_level->bip == BIP_DRP_FIFO_LEN)
    {
        (fifo_level->bips_lost)++; /* Added for monitoring (RC 11/17/94) */
        if(!bip_fifo_full)
        {
            bip_fifo_full = 1;
            logmsg("wrt_fifos: BIP FIFO full, data not being written");
        }
    }
    else
    {
        if(bip_fifo_full)
        {
            bip_fifo_full = 0;
            logmsg("wrt_fifos: BIP FIFO data being written again");
        }
        bip_fifo_idx = (++bip_fifo_idx) % BIP_DRP_FIFO_LEN;
        bip_rec = (bip_oto_rec_t *) (bips + (sizeof(bip_oto_rec_t) *
            bip_fifo_idx));
        (fifo_level->bip)++;
    }
}

return;

```

}
/***** /

Appendix T Listing of snd_bips.c File

```

/* "(@#)snd_bips.c 1.6" */
/*****
*
* ORNL I&C Division
* Module name:  snd_bips
* Version: 1.6
* Modified: 9/12/95 12:44:11
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
*             Oak Ridge National Laboratory
*             Instrumentation & Controls Division
*             Real Time Computer Systems Group
*             P O Box 2008, Bldg 3500, MS 6007
*             Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FUNCTIONAL DESCRIPTION:
*     Reads bips from shared memory and writes to DRP
*
*
* STATUS RETURN VALUES
*
*
*****/
#include "rec_spls.h"
#define BIP_DRP_PEER_NAME "_rec_spls"
#define BIP_DRP_HEADER "_bip_data.$"

void
snd_bips()
{
    static char      SccsID[] = "(@#)snd_bips.c 1.6 9/12/95 ORNL"; /* For ver
control */

    static struct timeval delay; /* Loop delay for checks */

    static char      drp_header[32];
    static char      peer_name[32];
    char             *sret;
    tcpio_status_t    status; /* DRP functions status */
    void              cshm();

    peer_name[0] = drp_header[0] = 0;
    bip_fifo_idx = 0;
    delay.tv_sec = 0;
    delay.tv_usec = BIP_WRT_SLEEP;

    /*
     * Create shared memory
     */
    (void) cshm();
    fifo_level->bip = 0; /* Initialize value (RC 11/22/94) */

    sret = strcpy(drp_header, bfname_p);
    sret = strcpy(peer_name, bfname_p);
    sret = strcat(drp_header, BIP_DRP_HEADER);
    sret = strcat(peer_name, BIP_DRP_PEER_NAME);
    if (strcmp(bfname_p, "bfb") == 0)
        sret = strcat(drp_header, "b");
    else
        sret = strcat(drp_header, "a");
    logmsg("perr name = %s, header = %s", peer_name, drp_header);

    /*

```

```

    * Connect to DRP
    */
status = tcpio_drp_connect(getenv("REC_SPLS_BIP_DRP_NODE"), peer_name, NULL);
if (status != TCPIO_OK)
{
    logmsg("snd_bips: Error on tcp_drp_connect");
    if (debug)
    {
        tcpio_print_status(status);
    }
    exit(-1);
}

/*
 * Main loop to write data when needed to drp
 */
for (;;)
{
    if (fifo_level->bip > 0)
    {
        status = tcpio_send_mssg(drp_header,
                                (mssg_size_t) sizeof(bip_oto_rec_t),
                                (bips + (bip_fifo_idx *
                                           sizeof(bip_oto_rec_t))),
                                (tcpio_flag_t) BLOCKING_IO);
        if (status != TCPIO_OK)
        {
            logmsg("snd_bips: tcpio error");
            if (debug)
            {
                tcpio_print_status(status);
            }
        }
        bip_fifo_idx = (++bip_fifo_idx) % BIP_DRP_FIFO_LEN;
        (fifo_level->bip)--;
    }
    else if (fifo_level->bip < 0)
    {
        exit(0);
    }
    else
    {
        if (select(0, (fd_set *) 0, (fd_set *) 0, (fd_set *) 0, &delay) < 0)
        {
            logerr("snd_bips: select error");
        }
    }
}
/* NOTREACHED */
return;
}
/*****

```

Appendix U Listing of snd_sotos.c File

```

/* "@(#)snd_sotos.c 1.6" */
/*****
*
* ORNL I&C Division
* Module name:  snd_sotos.c
* Version: 1.6
* Modified: 9/12/95 12:41:33
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
*           Oak Ridge National Laboratory
*           Instrumentation & Controls Division
*           Real Time Computer Systems Group
*           P O Box 2008, Bldg 3500, MS 6007
*           Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****
* FUNCTIONAL DESCRIPTION:
*   Reads sotos from shared memory and writes to DRP
*
* STATUS RETURN VALUES
*
*****/
#include "rec_spls.h"

#define SOTOS_DRP_PEER_NAME "_rec_spls"
#define SOTOS_DRP_HEADER "_oto_sti_data.$"
void
snd_sotos()
{
    static char      SccsID[] = "@(#)snd_sotos.c 1.6 9/12/95 ORNL"; /* For ver
control */

    static struct timeval delay; /* Loop delay for checks */
    void          cshm();
    char          drp_header[32];
    char          peer_name[32];
    tcpio_status_t status; /* DRP functions status */

    peer_name[0] = drp_header[0] = 0;
    soto_fifo_idx = 0;
    delay.tv_sec = 0;
    delay.tv_usec = SOTO_WRT_SLEEP;

    /*
     * Create shared memory
     */
    (void) cshm();
    fifo_level->soto = 0; /* Initialize value (RC 11/22/94) */

    (void) strcpy(peer_name, bfname_p);
    (void) strcpy(drp_header, bfname_p);
    (void) strcat(peer_name, SOTOS_DRP_PEER_NAME);
    (void) strcat(drp_header, SOTOS_DRP_HEADER);
    if (strcmp(bfname_p, "bfb") == 0)
        (void) strcat(drp_header, "b");
    else
        (void) strcat(drp_header, "a");

    /*
     * Connect to DRP
     */

```

```

status = tcpio_drp_connect(getenv("REC_SPLS_SOTO_DRP_NODE"), peer_name, NULL);
if (status != TCPIO_OK)
{
    (void) logmsg("snd_sotos: Error on tcp_drp_connect\n");
    if (debug)
    {
        tcpio_print_status(status);
    }
    exit(-1);
}

/*
 * Main loop to write data when needed to drp
 */
for (;;)
{
    if (fifo_level->soto > 0)
    {
        status = tcpio_send_mssg(drp_header,
                                (mssg_size_t) sizeof(sti_oto_rec_t),
                                sotos + (soto_fifo_idx * sizeof(sti_oto_rec_t)),
                                (tcpio_flag_t) BLOCKING_IO);
        if (status != TCPIO_OK)
        {
            logmsg("snd_otos: tcpio error");
            if (debug)
            {
                tcpio_print_status(status);
            }
        }
        soto_fifo_idx = (++soto_fifo_idx) % SOTO_DRP_FIFO_LEN;
        (fifo_level->soto)--;
    }
    else if (fifo_level->soto < 0)
    {
        exit(0);
    }
    else
    {
        if (select(0, (fd_set *) 0, (fd_set *) 0, (fd_set *) 0, &delay) < 0)
        {
            logerr("snd_sotos: select error");
        }
    }
}
/* NOTREACHED */
return;
}
/*****

```

Appendix V Listing of wrt_scnb.c File

```

/* "@(#)wrt_scnb.c 1.5" */
/*****
*
* ORNL I&C Division
* Module name: wrt_scnb
* Version: 1.5
* Modified: 6/26/95 10:41:33
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
* Oak Ridge National Laboratory
* Instrumentation & Controls Division
* Real Time Computer Systems Group
* P O Box 2008, Bldg 3500, MS 6007
* Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FUNCTIONAL DESCRIPTION:
* Reads STI CNBs from shared memory and writes to DRP
*
* STATUS RETURN VALUES
* NONE
*
*****/
#include "rec_spls.h"
#define DEF_CNB_STI_PATH "/run/data/cnb_sti.dat"
#define SCNB_MODE 0666 /* Mode for SCNB file */

void
wrt_scnb()
{
    static char ScnsID[] = "@(#)wrt_scnb.c 1.5 6/26/95 ORNL"; /* For ver
control */

    static struct timeval delay; /* Loop delay for checks */
    void cshm();
    char *epathp;
    int ffd; /* Flag file descriptor */
    int full; /* File system full flag */
    int n; /* Return value from write */
    int old_store; /* Store flag last time */
    char path[64];
    int sfd; /* STI file descriptor */
    char *sret;

    delay.tv_sec = 0;
    delay.tv_usec = SCNB_WRT_SLEEP;
    full = 0;
    scnb_fifo_idx = 0;

    /*
     * Create shared memory
     */
    (void) cshm();
    fifo_level->scnb = 0; /* Initialize value (RC 11/22/94) */

    /*
     * Main loop to write data when needed to disk
     */
    old_store = 0;
    logmsg("starting wrt_scnb main loop");
    for (;;)
    {
        if (*store != old_store)

```

```

{
    if (*store)
    {
        /*
         * Open STI file
         */
        if ((epathp = getenv("CNB_STI_PATH")) != NULL)
            sret = strcpy(path, epathp);
        else
            sret = strcpy(path, DEF_CNB_STI_PATH);
        if ((sfd = open(path, O_WRONLY | O_CREAT | O_TRUNC, SCNB_MODE))
            == -1)
        {
            logerr("wrt_scnb: STI file open error");
            *store = 0;
        }
        else
        {
            full = 0;          /* Assume space available at COMEX */

            /*
             * Create file that tells integration that data is being stored
             */
            if ((ffd = open(sstis_path, O_RDWR | O_CREAT, 0666)) < 0)
                logerr("Error opening STORING_STIS flag file");
            (void) close(ffd);
        }
    }
    else
    {
        (void) close(sfd);      /* Close data file */
        (void) unlink(sstis_path); /* Remove flag file */
    }
    old_store = *store;
}

if (fifo_level->scnb > 0)
{
    if (*store && (!full))
    {
        if (n = write(sfd, scnb + (scnb_fifo_idx * scnb_wrt_len),
            scnb_wrt_len) < 0)
        {
            if (ENOSPC == errno)
            {
                full = 1;
                logerr("wrt_scnb: STI CNB disk full");
            }
            logerr("wrt_scnb: sti beam write error");
        }
    }
    if (++scnb_fifo_idx == SCNB_DSK_FIFO_LEN)
        scnb_fifo_idx = 0;
    (fifo_level->scnb)--;
}
else if (fifo_level->scnb < 0)
{
    exit(0);
}
else
{
    if (select(0, (fd_set *) 0, (fd_set *) 0, (fd_set *) 0, &delay) < 0)
    {
        logerr("wrt_scnb: select error");
    }
}

```

```
    }  
  }  
}  
/* NOTREACHED */  
return;  
}  
/*****/
```

Appendix W Listing of end_process.c File

```

/* "@(#)end_process.c      1.5" */
/*****
*
*                                     ORNL I&C Division
* Module name:  end_process.c
* Version: 1.5
* Modified: 11/21/95 11:59:48
*
* Project: AMFIP-II
*
* Created by: Bobby R. Whitus
*            Oak Ridge National Laboratory
*            Instrumentation & Controls Division
*            Real Time Computer Systems Group
*            P O Box 2008, Bldg 3500, MS 6007
*            Oak Ridge, TN 37831-6007 (615) 574-8673
*
*****/
* FUNCTIONAL DESCRIPTION:
*   Clean up and exit
*
*
* STATUS RETURN VALUES
*   Never returns
*
*****/
#include "rec_spls.h"

void
end_process()
{
    static char    SccsID[] = "@(#)end_process.c      1.5 11/21/95 ORNL"; /* For
ver control */

    int            wait_stat;      /* Wait status */
    int            wait_ret;      /* Return value for wait function */
    struct shmid_ds *null_ptr = NULL;
    int            i;
    logmsg("process exiting");

    (void) sleep(2);              /* Give some time for FIFOs to empty */

    /*
    * Tell child processes to die, wait for each to expire
    */
    fifo_level->scnb = -1;
    fifo_level->soto = -1;
    fifo_level->bip = -1;
    for (i = 0; i < num_children; i++)
    {
        wait_ret = wait(&wait_stat);
    }

    /*
    * Remove shared memory, detach then remove
    */
    if (shmdt((char *) fifo_level) < 0)
        logerr("shmdt error for fifo level");
    if (shmdt((char *) scnb) < 0)
        logerr("shmdt error for scnb");
    if (shmdt((char *) sotos) < 0)
        logerr("shmdt error for sti otos");
    if (shmdt((char *) bips) < 0)
        logerr("shmdt error for bips ");

    /*

```

```

    * The shared memory for fifo_level cannot be removed because it is
    * also used by track_monitor to monitor BIP and STI OTOs that are lost
    */
    if (shmctl(scnbsmid, IPC_RMID, null_ptr) < 0)
        logerr("shmctl error for scnbsmid");
    if (shmctl(sotosmid, IPC_RMID, null_ptr) < 0)
        logerr("shmctl error for sotosmid");
    if (shmctl(bipsmid, IPC_RMID, null_ptr) < 0)
        logerr("shmctl error for bipsmid");

/*
 * Close files and exit
 */
    for (i = 0; i < INPUT_BUFNUM; i++)
        if (aiocancel(&(aio_result[i])) < 0)
            logerr("aiocancel %d error", i);

    (void) unlink(rdy_path);
    (void) unlink(sstis_path);

    if (ioctl(hfd, HIP_CLOSE, 0) < 0)
        logerr("Ioctl error on HIP_CLOSE");

    (void) close(hfd);
    exit(0);
}
/*****

```

Vita

Bobby R. Whitus graduated from Central High School in Wartburg, Tennessee in 1972.

One year later he completed training as an electronics technician from Central High Electronics School, Wartburg, Tennessee with a 4.0 GPA. In 1973 he began employment with EG&G ORTEC, Inc. (ORTEC) as an electronics technician. Work activities at ORTEC included printed circuit layout and support of nuclear instrument design as a member of the timing and optics R&D group. From 1974 until 1985 he was employed as an instrument mechanic at Martin Marietta Energy Systems, Inc. (MMES) in Oak Ridge, Tennessee. Activities performed as an instrument mechanic included repair and calibration of electronic instruments, computer system maintenance and electronic design. An Associate of Science degree in Pre-Engineering from Roane State Community in Harriman, Tennessee was earned while employed at MMES. A BSEE degree from the University of Tennessee with a 3.9 GPA was earned in 1987.

Bobby has been employed by the Oak Ridge National Laboratory from 1988 to present as a development engineer. His present capabilities include system and network management, software development, hardware design and project management. His accomplishments as development engineer include development of communications software for a large digital signal processing system, design of several local area networks, completion of two publications and leadership of a team that developed an industrial automation system.