Implementation of the Arctic Network Control Interface System

by

Christopher W. Ward

Submitted to the Department of Electrical Engineering and Computer Science in

Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Computer Science and Engineering and Master of

Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 28, 1996

Copyright 1996 Christopher W. Ward. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.

Author .....
Department of Electrical Engineering and Computer Science
May 28, 1996

Certified by .....
George A. Boughton
Thesis Supervisor

Accepted by .....
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUN 11 1996

LIBRARIES

Eng.

Implementation of the Arctic Network Control Interface System

by

Christopher W. Ward

Submitted to the Department of Electrical Engineering and Computer Science

May 28, 1996

In Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The Arctic Network Control Interface System (ANCIS) is a software system used to interface to the Arctic Switch Fabric Network, the interconnection network of the StarT-Voyager. ANCIS provides a relatively simple interface to access the registers of the maintenance interface of each Arctic router in the network. Access to these registers enables testing, configuration, and access to statistics of the Arctic Switch Fabric Network. This thesis describes the implementation and operation of ANCIS.

Thesis Supervisor: George A. Boughton
Title: Principal Research Engineer, MIT Laboratory for Computer Science

Table of Contents

List of Figures

Chapter 1

Introduction

The Arctic Network Control Interface System (ANCIS) is a software system designed to interface to the Arctic Switch Fabric Network, the interconnection network of the StarT-Voyager. ANCIS provides a relatively simple interface to access the registers of the maintenance interface of each Arctic router in the network. Access to these registers enables testing, configuration, and access to statistics of the Arctic Switch Fabric Network.

While the design of the ANCIS architecture had been completed previously, only parts of the system had been implemented. My task was to complete the implementation of the Arctic Network Control Interface System. This included filling in the remaining gaps in the provided system framework by implementing functions at all levels of the software system. The project also involved testing and debugging the entire system, including the hardware interface.

1.1 StarT-Voyager Project

StarT-Voyager is message-passing parallel computer with support for distributed cache-coherent shared memory being developed by the Computation Structures Group at MIT. StarT-Voyager has 32 nodes, and according to the paradigm of the StarT-Voyager, each of its nodes may pass messages to any other node.

These messages are delivered by the Arctic Switch Fabric Network, a 32 leaf fat tree composed of 64 Arctic routers.[1] An Arctic router is a 4x4 packet switch on a chip.[1] Four Arctics are grouped together on each circuit board, and The Arctic Switch Fabric Network consists of 16 of these boards.[3]

1.2 Arctic Maintenance Interface

Arctic has a maintenance interface through which internal data may be accessed and the

chip may be controlled and tested. These functions are provided through access operations which read and write the Arctic maintenance registers. The Arctic maintenance interface is a JTAG interface; it (in most ways) follows the IEEE JTAG standard (IEEE Std. 1149.1).[1] This JTAG interface consists of four signals and a Test Access Port (TAP) Controller. The four signals of the interface are Test Clock (TCK), Test Data In (TDI), Test Data Out (TDO), and Test Mode Select (TMS). The TAP controller is a finite state machine. TAP state transitions are determined by the TMS signal. By stepping through a specific sequence of states, the TAP controller can shift bits from the TDI line into a particular register or shift bits out of a register onto the TDO line. In this way Arctic maintenance registers can be written and read. With a sequence of writes and reads of instruction and data registers the Arctic access operations can be executed.

The maintenance (JTAG) interfaces of all 64 Arctics in the Switch Fabric Network are accessed through only four signals. This is done by connecting the JTAG ports of the Arctics to form one scan ring. The TDO signal of one Arctic is connected to the TDI of the next one so that data can be shifted into and out of every Arctic. The Test Clock and TMS signals are broadcast to every Arctic at the same time. So all the Arctics are always in the same TAP state; only the data differs from one chip to the next. This configuration allows complete access to individual Arctics with only one JTAG interface to the Arctic Switch Fabric Network.

This single JTAG interface to the network is controlled by the JTAG controller board. This hardware acts as the interface between the computer running ANCIS and the JTAG interface to the Arctic Switch Fabric Network. The JTAG controller board receives bits from the TDO signal from the network, buffers these bits and makes them available to ANCIS through the ISA bus of the computer. The controller board also receives TDI and

TMS bits from ANCIS through the ISA bus, buffers these bits, and sends them into the Arctic network.

1.3 ANCIS

ANCIS is a software system which runs on a personal computer and accesses the JTAG interface of the Arctic Switch Fabric Network through the JTAG controller board on the ISA bus of the computer. ANCIS not only controls the JTAG interface, but also masks the details of the necessary protocols, allowing a user or the StarT-Voyager to access the Arctics using a relatively simple script interface. This abstraction is achieved through a system architecture with several layers, each hiding an interface protocol.

The ANCIS architecture has three main modules: the script interface, the Arctic interface, and the JTAG controller board interface. The script interface reads and parses the script files, breaking them down into operations for the Arctic interface. The Arctic interface translates these high level commands into the bit stream to be sent to the Arctic network through the JTAG scan ring. Finally the JTAG controller board interface sends the bit stream to the JTAG controller board. This module handles the controller board protocol, sending commands and monitoring status

The Arctic interface module divides further into four layers of abstraction. These four levels correspond to sets of procedures supported by Arctic. The upper level procedures are accessible by the script interface and in turn call the Arctic access operations described above. Each Arctic access operation breaks down into access macros, each of which call JTAG transaction macros. The JTAG transaction macros each consist of a sequence of JTAG TAP controller states.

The boundary scan and low level scan functions do not follow the same layers of abstraction. The Extest, Arcintest, and Low Level Scan operations directly call a separate

set of macros because these functions are very different from the rest of the ANCIS functions. Extest is a test external to the chip for testing the board. Arcintest is an internal test of the Arctic. Low Level Scan accesses the manufacturing test rings. Currently only Arcintest has been implemented.

1.4 The Project

My project was to complete the implementation of ANCIS, including testing and debugging the entire system. I was also partially responsible for the JTAG controller board. This responsibility included determining the proper characteristics of the board and verifying that the board would provide a suitable interface to ANCIS. A commercially produced board, the Corelis PC-1149.1/100F High Speed PC-AT Bus Boundary-Scan Controller Board, was eventually selected. At this point my task was to develop software to use the Corelis board and to integrate the provided software into ANCIS. The Corelis hardware operates the JTAG interface using slightly different methods than the hardware for which ANCIS was originally designed. As a result, much of the lower level software needed to be rewritten to use the Corelis board.

The ANCIS structure had been designed previously, and some of the system had already been implemented. However there were still many functions which need to be written and tested and features that needed to be added. I implemented most of the high level functions, designed and implemented the arcintest functions, integrated the low level Corelis software, and modified many functions throughout ANCIS. I also debugged the previously existing software as well as the final system.

Chapter 2

Background

2.1 Arctic Maintenance Interface

2.1.1 JTAG Boundary Scan Standard

The maintenance interface of Arctic is a standard JTAG interface as set forth by IEEE Std. 1149.1. A JTAG interface consists of a set of signals and a Test Access Port (TAP) controller. The standard signals are Test Mode Select (TMS), Test Data In (TDI), Test Data Out (TDO), a Test Clock (TCK), and an optional Reset signal (TRST).

The TAP controller is a state machine whose state is determined by input from the TMS signal. Refer to the state diagram in Figure 2.1. Each rising edge of TCK causes a state transition based on the value of TMS. A few important states to notice are Run-Test/Idle, Shift-DR, and Shift-IR. Run-Test/Idle is the base state for ANCIS. All ANCIS operations eventually return the TAP controller to Run-Test/Idle. In the Shift-DR state, each TCK shifts a bit from TDI into the selected data register and a bit out to TDO. The selected data register depends on the current instruction. Shift-IR is analogous to Shift-DR for the instruction register.

The TMS signal determines these state transitions. The TDI line carries the data to be shifted into the JTAG interface, and the TDO signal is the output data. TMS, TDI, and TDO are serial data lines. One bit for each of these signals is processed by the JTAG interface for each TCK clock cycle. TRST is an optional signal which places the TAP controller in the Test Logic Reset state when asserted. This signal is optional because the state machine is constructed such that a sequence of ones in the TMS signal (5 bits or more) produces the same result.

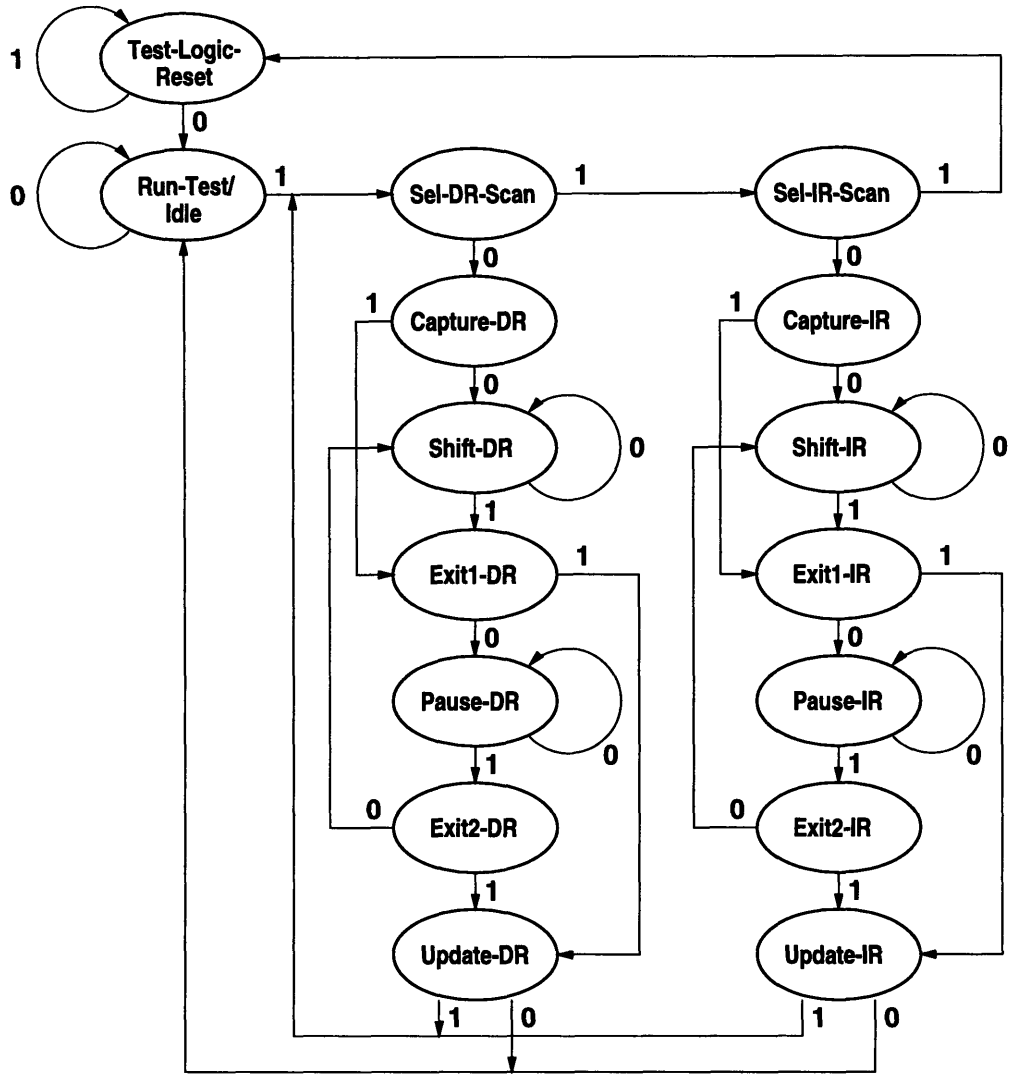

Figure 2.1: TAP Controller State Diagram

Data shifted into the JTAG interface is shifted into a selected register. This register is said to be between TDI and TDO because a continuous bit stream can be shifted from TDI through the selected register to TDO. There are several standard registers which may be selected. The JTAG Interface Instruction Register is selected when the TAP Controller is in an IR state. The value in this instruction register selects a particular data register when the TAP Controller is in a DR state. The two standard data registers are the Bypass register

and the Boundary Scan Cells. The Bypass register is a one bit register used to simply pass data through the interface. The Boundary Scan Cells are a ring of cells, usually corresponding to pins of the chip, which form a register. The JTAG standard also allows for extra test specific data registers.

2.1.2 The Arctic JTAG Interface

The Arctic JTAG interface includes several implementation details which are not covered by the discussion of the JTAG standard. First the standard allows for additional registers within the maintenance interface. Arctic has several of these test specific registers which may be selected by loading a particular value into the JTAG Interface Instruction Register. The Arctic Mode Register controls the Arctic's mode of operation. The Status and Data Register allows access to Arctic registers. The Status part is two bits and contains the most recent status of the read or write. The Data part is selected from many Arctic registers including the statistics registers, the control register, the configuration registers, and the error and buffer free counters. The Access Operation Address Register is used to select a particular Arctic register for access. The Access Operation Instruction Register determines the type of operation, read or write, to be performed on the Status and Data Register. Finally, the Low Level Mode Scan Register is used to scan data through Arctic's internal scan cells.

A second Arctic feature is that the Arctic network uses one JTAG interface to access the maintenance interfaces of all 64 Arctic chips. The Arctics are arranged in a ring. The TDO signal of the first chip is connected to the TDI of the next, etc. So data in the TDI data stream can be shifted through several chips and into a target register. Similarly data can be shifted out through several chips and back to ANCIS. The TMS signal is broadcast to all the chips at once, so all the Arctics are always in the same TAP state. See Figure 2.2. Although each Arctic is the same TAP state, different data may be shifted into each chip,

so some Arctics may be accessed while others are bypassed. This is done by shifting the bypass instruction into the instruction registers of those Arctics which are not to be accessed for a particular operation. An Arctic which is bypassed shifts data into and out of a one bit bypass register rather than a data register.

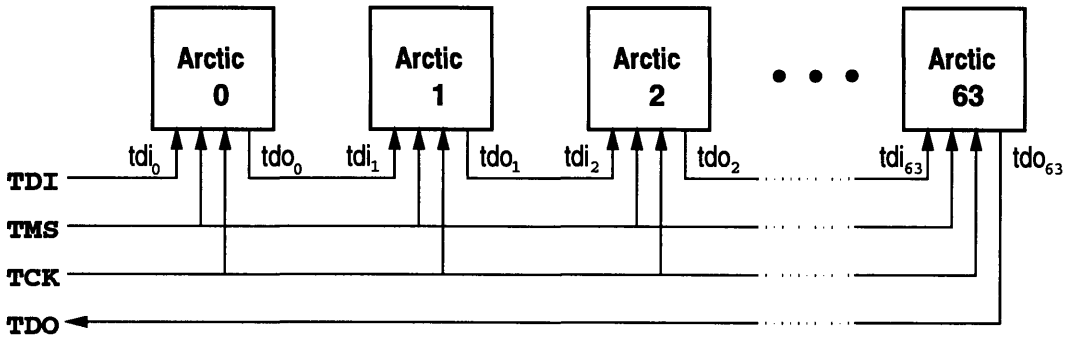

Figure 2.2: Arctic Scan Ring

The Arctic maintenance interface actually differs from IEEE Std. 1149.1 on several details concerning some of the boundary scan operations. These differences are the result of several facts. First, most Arctic data lines receive a new bit every half clock cycle, while the standard only handles a complete clock cycle. This necessitates many pins having two associated scan cells. More problems are that Arctic receives inputs from multiple clock domains and that the control registers can only be loaded through the maintenance interface. For details on these differences and on the Arctic maintenance interface in general, see the *DRAFT Arctic User's Manual.*[1]

2.2 ANCIS Structure

The structure of ANCIS was laid out by Elizabeth Ogston in her thesis, *The Arctic Switch Fabric Control Interface System.*[3] This structure follows a functional hierarchy described in the *DRAFT Arctic User's Manual.*[1] This structure allows multiple levels of abstraction, masking several layers of protocol from the user. The result is an interface to

ANCIS through a relatively simple script file or alternately through the use of high level library functions.

2.2.1 Script Interface

The script interface consists of the functions in the file "scripts.c" (Appendix D.5 on page 86). This is the highest level interface, interacting most directly with the user. The script interface provides a relatively simple way to use ANCIS functions through a script file of a particular format. The script interface reads this scripting language and extracts the commands, data, and other parameters. With this information, the script interface then invokes the proper ANCIS functions.

2.2.2 Upper Level

The upper level is the top level of the arctic interface and consists of the functions in the file "upper_level.c" (Appendix D.7 on page 113). Each Arctic operation which can be invoked from the script interface is represented by a function in the upper level. This includes reads and writes of Arctic registers, a controller board reset, and Arcintest. In this level, register operations and addresses are determined. Also, the data length is specified since each data register may have a different length.

2.2.3 Arctic Access Operations

Arctic Access Operations, made up of the functions in the file "arctic_access_ops.c" (Appendix D.9 on page 129), are used to read and write Arctic registers. Accessing these registers is a two part operation, and some registers require a check of the status of the operation. Checking status and re-reading the data if necessary is done at this level.

2.2.4 Access Macros

Access Macros provide functions, located in the file "access_macs.c" (Appendix D.11 on page 134), to execute the steps necessary to read or write a particular register. These functions allow writing to the address and instruction registers in order to select the cor-

rect data register. They also provide the ability to write data to and read data and status from the selected data register.

2.2.5 JTAG Transaction Macros

The functions in the file "jtag_trans.c" (Appendices D.13 and D.14) are the JTAG transaction macros. These functions create the final bit streams to be sent to the JTAG interface. At this level, bypass bits are inserted for any bypassed Arctics before sending input data and removed before returning output data.

2.2.6 Board Level

The board level functions in the file "board.c" (Appendices D.16 and D.18), provide an interface to the hardware, the Corelis JTAG board which sends the signals to the Arctic maintenance interface. This level handles all of the hardware concerns. These functions are a combination of Corelis supplied drivers and higher level functions based on Corelis test software.

2.3 Corelis JTAG Board and Software

Part of the task of implementing ANCIS was setting requirements for and choosing the hardware to send the signals to the maintenance interface. A board had been designed previously in house that would connect to the ISA bus of a PC. However there were some concerns about this board, and it was decided that a commercially produced board would be purchased, rather than modify the existing board. Using a commercially produced board has the advantage of minimizing development time and problems. The Corelis PC-1149.1/100F High Speed PC-AT Bus Boundary-Scan Controller Board was identified as a possibility, and my task was to verify that its hardware and software was appropriate for the application. The Corelis board met the requirements and was purchased.

The Corelis JTAG Controller board is based on the SCANPSC100F Parallel/Serial Converter.[2] This chip is controlled by writing values to its internal registers. Writes to

these registers control mode and configuration such as TCK frequency and internal counter control. Reads and writes to these registers also allow access to the TMS, TDI, and TDO signals. Two of these chips are on the Corelis board allowing four distinct JTAG interfaces.

In addition to the two SCANPSC100F Parallel/Serial Converters, the Corelis board also includes two FIFOs to buffer the TDO and TDI data streams when "turbo" mode is selected. RS-422 drivers and a line receiver allow one JTAG port to use RS-422 differential signals. Also, the on-board SCAN18374T provides a test JTAG target for self-testing. The SCAN18374T has a Test Access Port which can be connected, in software, to the board's JTAG interface. For more details on the Corelis board, refer to the associated user's manual.[2]

The software included with the Corelis board provides C language source code for a self-test program. This source code was meant as an example for writing code for the board. It provides many functions which perform basic tasks with the board. These low level functions, as well as some of the higher level functions, are used directly in ANCIS. Other ANCIS board level functions are based heavily on code provided with the Corelis board. See Appendix C for documentation on the provided software.

Chapter 3

Implementation

3.1 Main Functions

3.1.1 Script Interface

The top level of ANCIS is the script interface. This is the entry point of the system. ANCIS reads a script file which lists the commands to be run along with input data and other parameters. The script interface is responsible for interpreting the script file, extracting the commands and parameters, and then calling functions in the upper level of the Arctic interface to execute those commands.

The script interface is implemented by reading a line of the script file, interpreting the line, calling an appropriate function, and then reading a new line. This cycle is repeated until the end of the script file is reached. When a line from the script file is recognized as an Arctic command, a function specific to that command is called. This function will read further data and parameters from the script file if necessary and call the appropriate function in the upper level, passing along any input data and parameters.

The comment character for script files is '#'. Lines beginning with this character as well as blank lines are ignored as long as such a line falls between Arctic commands. However, once an Arctic command is recognized, the script interface expects a particular format. This format varies slightly from one command to the next because the required parameters vary. The format for each command is described below. For more detailed information on the script interface, refer to the source code file "scripts.c" in Appendix D.5 on page 86.

The first command is likely to be **start_log**. This command opens a log file and writes selected information to the file as ANCIS runs. The **start_log** command should be fol-

lowed by the name of the log file on the same line. Another parameter may optionally follow the log file name, again on the same line. This optional parameter is the log level, ranging from 1 to 7. If the parameter is omitted, the log level assumes the default value of 1. The log level is used to vary the type of information logged. A level of 7 requests information from functions at all levels of ANCIS. This is used primarily for debugging. A level of 1 is used for normal operation, only logging errors and high level information such as values read and written to registers.

```
start_log test1.log 7
```

The **end_log** command closes a log file previously opened by the **start_log** command. Only one log file may be open at one time, so no parameters are needed with the command. The following example also illustrates the use of comments.

```
# These two lines are comments. The next
# line is also ignored because it is blank.

end_log
```

The **reset_controller** command is also simple to use, taking no parameters. This command invokes a reset of the JTAG controller board.

```
reset_controller
```

The remaining commands are Arctic operations and require at least one additional parameter. The script file should indicate which Arctics are intended for the operation. The remaining Arctics will be bypassed. This is done in one of two ways. If no input data is required for the command, the Arctics are specified by listing each on a separate line following the command.

```
example_command
ARCTIC 0
ARCTIC 1
ARCTIC 22
ARCTIC 63
```

If input data is required for the command, the Arctics are listed as before, but the data for each is listed on the line following the Arctic.

```
input_data_command
ARCTIC 0
data
ARCTIC 1
data
ARCTIC 22
data
ARCTIC 63
data
```

The **reset** command requires only the Arctic list. This command resets all specified Arctics.

```
reset
ARCTIC 0
ARCTIC 1
```

The **read_psc** command takes the Arctic list. This command reads the 18 bit control register and displays the resulting data stream.

```
read_psc
ARCTIC 1
ARCTIC 2
```

The command **write_psc** writes data to the 18 bit control register of all specified Arctics. It requires a list of Arctics with input data for each. The data should be in the format of five hexadecimal digits. The two far left bits are extra and will be ignored.

```
write_psc
ARCTIC 0
12345
ARCTIC 3
89abc
```

The command **clear_stats** requires only the list of Arctics. This command clears all of the statistics registers for all four ports of the specified Arctics.

```
clear_stats
ARCTIC 0
```

The **all_stats** command reads all of the statistics registers (36 bits + 2 status bits) for all four ports of the specified Arctics and can optionally clear these registers after reading them. This command takes an extra parameter on the line following the command. If the statistics registers should be cleared after the read, then the first word on this line should be "clear" (any capitalization). Otherwise, anything else may be on the line, but it should not be left blank. The list of arctics then follows on the next line.

```
all_stats
NO-clear
ARCTIC 0
ARCTIC 1
```

The **port_stats** command is identical to the **all_stats** command described above except that it reads statistics from only one specified port (0-3). The extra parameter specifying the port number should be on the same line as the clear parameter and should come first on the line. The two parameters should be separated only by spaces (no commas). Since there is already a parameter on the line, it will not be blank, so the clear parameter may be omitted. (In that case the registers would not be cleared.)

```
port_stats
0 Clear
ARCTIC 0
```

The **port_stat** command is identical to the **port_stats** command described above except that it reads only one statistics register. The extra parameter specifying the specific statistic register should be on the same line as both the port and clear parameters and should come second on the line. The statistic register to be read should be chosen from the following: "PACKETS", "PRIORITY", "UP", "DOWN", "IDLE", and "WAIT". Again the clear parameter may be omitted.

```
port_stat
0 IDLE Clear
ARCTIC 0
```

The command **errors** reads the 60 bit error counters of the specified Arctics and displays the returned stream of bits. The clear parameter follows on the next line after the command, and the list of Arctics then follows, just as in the format of the **all_stats** command above. Note that the line following the command, the line for the clear parameter, should not be left blank.

```
errors
Do_NOT_Clear
Arctic 0
```

The **bufs_free** command reads the value of the 20 bit buffer free counter. This command takes only an Arctic list.

```
bufs_free
ARCTIC 0
```

The **change_mode** command is used to write the Arctic mode register. The mode should be specified on the line following the command. The available modes are "normal", "config" for configuration mode, and "low_level_test".

```
change_mode
config
ARCTIC 0
```

The **cfg_write_all** command writes data to the configuration registers of all the Arctic ports. Each port has five 32 bit configuration registers. With this command the same data is broadcast to each port. The script file should specify all of the parameters for each of the five registers in order. Each register requires a list of Arctics and associated data (8 hexadecimal digits). Even though the list of Arctics is given for each register, this list should be the same for each. One extra line should precede each list of Arctics and data in the script

file. This line will be ignored, so it may be left blank or used as a comment or a reminder of the associated register as in the example below.

```
cfg_write_all
reg1
ARCTIC 0
020c4100
ARCTIC 1
020c4100
reg2
ARCTIC 0
08041cc5
ARCTIC 1
00000000
reg3
ARCTIC 0
f0001000
ARCTIC 1
01234567
reg4
ARCTIC 0
f0002000
ARCTIC 1
89abcdef
reg5
ARCTIC 0
000300ff
ARCTIC 1
000300ff
```

The **cfg_write_port** command is the same as the **cfg_write_all** command above except that it only writes to the configuration registers of one port (0-3). It also requires one extra parameter, the port number. This parameter should be put on the line following the command. The remaining lines follow the rules above for **cfg_write_all**.

```
cfg_write_port
0
reg1
ARCTIC 0
0ffbdff8
reg2
ARCTIC 0
040f7bfe
```

```
reg3
ARCTIC 0
00000000
reg4
ARCTIC 0
00000000
reg5
ARCTIC 0
000300ff
```

The **cfg_read_port** command reads the values of the five configuration registers of one Arctic port (0-3). It reads 32 data bits and 2 status bits for each of five registers. This command requires that a port number be specified on the line following the command. A list of Arctics should follow, starting on the next line.

```
cfg_read_port
3
ARCTIC 10
ARCTIC 11
ARCTIC 15
```

The **arcintest** command is used to perform a boundary scan function. Because there is a large amount of data, separate files are used for reading and writing the test data. (See Appendix B on page 60 for details on the file formats.) The names of both data files should be on the line following the **arcintest** command. The input file name should be first, followed by the output file name. The two file names should be separated only by spaces. Next should be the list of Arctics. The arcintest() function is treated separately from the main body of Arctic operations. See section 3.2 for information about the arcintest boundary scan operation.

```
arcintest
in.dat out.dat
ARCTIC 0
ARCTIC 1
```

See section 4.3 for some example script files.

3.1.2 Upper Level Functions

These functions and most of the other functions take an argument called *arctics*. This argument contains information indicating the Arctics on which to perform the given function. The variable *arctics* is an array of integers with as many elements as the total number of Arctics. Each element is associated with one Arctic. If an element is nonzero, the associated Arctic is to be included in the operation. Otherwise the element's value is zero, and the Arctic should be bypassed. For more detailed information on these functions, refer to the source file "upper_level.c" in Appendix D.7 on page 113.

These functions and many others take an argument that is a data abstraction called *data_stream*. ANCIS uses this data structure to pass data streams or bit streams between functions. All data meant for TMS, TDI, or TDO signals is encapsulated in this structure type. The functions used for operating on the *data_stream* type are in the file "data_stream.c". See Appendix D.3 on page 74.

void read_psc(struct data_stream *data, int arctics[]);

The function read_psc() reads the 18 bit value of the control register. The value read is returned in *data*.

void write_psc(struct data_stream *data, int arctics[]);

write_psc() writes the value of the argument *data* to the 18 bit control register.

void reset(int arctics[]);

The function reset() writes a one bit value to the reset registers of the Arctics specified by the argument *arctics[]*. This causes a reset of the specified Arctics.

void reset_controller();

reset_controller() resets the JTAG controller board.

void port_stat(int port, enum STAT stat, struct data_stream *data, int clear, int arctics[]);

port_stat() reads the *stat* statistic of the *port* port (0-3). The 38 bit value (36 bits of data, + 2 bit status) is returned in *data*. If *clear* is equal to one, the register is cleared following the read. Otherwise *clear* should be zero.

void port_stats(int port, struct data_stream *data[6], int clear, int arctics[]);

The function port_stats() reads all six of the statistics of the *port* port. The six 38 bit values (36 bits of data, + 2 bit status) are returned in *data[6]*. If *clear* is equal to one, the registers are cleared following the read. Otherwise *clear* should be zero.

void all_stats(struct data_stream *data[4][6], int clear, int arctics[]);

all_stats() reads all of the statistics registers. The data is returned in *data[4][6]*. If *clear* is equal to one, the registers are cleared following the read. Otherwise *clear* should be zero.

void clear_stats(int arctics[]);

clear_stats() clears all of the statistics registers.

void errors(struct data_stream *data, int clear, int arctics[]);

errors() reads the error counters. The 60 bit value is returned in *data*. If *clear* is equal to one, the error counters are cleared following the read. Otherwise *clear* should be zero.

void bufs_free(struct data_stream *data, int arctics[]);

bufs_free() reads the buffer free counters. The 20 bit value is returned in *data*.

void cfg_read_port(int port, struct data_stream *data[5], int arctics[]);

The function cfg_read_port() reads the configuration registers of port number *port*. The five 34 bit values (32 bits of data, + 2 bit status) are returned in *data[5]*.

void cfg_write_port(int port, struct data_stream *data[5], int arctics[]);

cfg_write_port() writes the configuration registers of port number *port* (0-3) with the data in *data[5]*.

void cfg_write_all(struct data_stream *data[5], int arctics[]);

cfg_write_all() writes the configuration registers of all four ports with the data in *data[5]*.

void change_mode(enum MODE mode, int arctics[]);

change_mode writes the Arctic mode register with value *mode*.

3.1.3 Arctic Access Operations

Arctic Access Operations perform the split phase operations of reading and writing data registers. These functions allow the selection of a particular register and the subse-

quent read or write of the associated data. Refer to the source file "arctic_access_ops.c" in Appendix D.9 on page 129 for more information on the use of these functions.

void read_reg (enum REGISTERS address, int arctics[]);

read_reg() selects the data register at *address* for a read so that a subsequent call to get_status() will return the value of the register.

void write_reg (enum REGISTERS address, struct data_stream *ds, int data_len, int arctics[]);

write_reg() writes the data in *ds* to the register at *address*. For some registers, a subsequent call to get_status() is necessary to ensure that the write has completed.

int get_status (struct data_stream *ds, int data_len, int arctics[], int status);

get_status() repeatedly reads the status and data register until the status indicates a completed operation (or an error). If the *status* argument is 0, the data register will be read, but the status will not be checked. This is used for registers which complete quickly and do not need status.

3.1.4 Access Macros

The Access Macros are used to access the Arctic JTAG registers. For details on these functions, refer to the source code in the file "access_macs.c" in Appendix D.11 on page 134.

void write_AOAreg(enum REGISTERS address, int arctics[]);

write_AOAreg writes *address* to the Access Operation Address Register.

void write_AOSandDreg(struct data_stream *ds, int arctics[], int data_len);

write_AOSandDreg writes the data *ds* to the status and data register. The integer *data_len* is the length of the data for each Arctic. This length should include status bits if needed for the particular register.

void write_AOIreg(enum INSTRS instruction, int arctics[]);

write_AOIreg writes *instruction* to the Access Operation Instruction Register.

void read_AOSandDreg(struct data_stream *ds, int arctics[], int data_len);

read_AOSandDreg reads the status and data register, placing the returned data in *ds*. The integer *data_len* is the length of the data for each Arctic.

void write_Mreg(short mode, int arctics[]);

write_Mreg writes *mode* to the Arctic Mode Register.

3.1.5 JTAG Transaction Macros

The JTAG transaction macros handle the preparation of the data stream before it is sent to the maintenance interface and again when it is returned. Specifically, functions at this level take care of bypassing any Arctics which have not been selected. This includes loading the bypass instruction and inserting and removing bypass bits in the data stream. These functions interact with the hardware interface functions. JTAG transaction macros have additional functionality when used with the Verilog model for testing. The paradigm for this testing scheme is different and requires extra work at the JTAG transaction level. (See section 4.1.1.) For detailed information on the operation of these functions, refer to the files "jtag_trans1.c" in Appendix D.13 on page 138 and "jtag_trans2.c" in Appendix D.14 on page 147. The source file "jtag_trans1.c" is for use with Verilog testing and "jtag_trans2.c" contains the code to be used instead when using the Corelis controller board.

void load_JTAGIR(enum JTAG_INST instr, enum JTAG_INST alternate, int arctics[]);

load_JTAGIR() loads the JTAG Interface Instruction Register with the instruction *instr* for all Arctics specified in the array *arctics[]*. The instruction *alternate* is loaded for all other Arctics. Returned data is ignored. This function will take the TAP controller through the following states:

Run-Test/Idle	(assumes this starting state)
Select-DR-Scan	
Select-IR-Scan	
Capture-IR	
Shift-IR	(will likely spend multiple cycles in this state)
Exit1-IR	
Update-IR	
Run-Test/Idle	

void load_JTAGDR(struct data_stream *ds, int arctics[], int data_len);

load_JTAGDR() loads the (previously) selected JTAG Data Register with *ds* and puts the returned data in *ds*. load_JTAGDR() inserts a bit into the data stream for each bypassed Arctic before sending the data. It also removes these bypass bits from the output data before returning it in *ds*. This function will take the TAP controller through the following states:

Run-Test/Idle	(assumes this starting state)
Select-DR-Scan	
Capture-DR	
Shift-DR	(will likely spend multiple cycles in this state)
Exit1-DR	
Update-DR	
Run-Test/Idle	

3.2 Arcintest

3.2.1 Description

The arcintest operation is a boundary scan function used to test the Arctic chip internally. Arcintest isolates the chip from its pins and shifts input data into the boundary scan cells. For each pattern of data shifted into the cells, the system clock is cycled, and the resulting scan cell values are shifted out as the next input pattern is shifted in. Only the boundary scan cells associated with input pins are specified in the input data, and all output data except that for scan cells associated with output pins are ignored. So by using arcintest to place test values on the Arctic input pins and examine resulting values on the output pins, the chip itself can be tested.

The arcintest operation can be invoked through the script interface using the **arcintest** command in the script file. See section 3.1.1 for more information on using the script interface.

3.2.2 Related Functions

Arcintest is executed with a call to the upper level function, arcintest(). This function then directly calls several special JTAG transaction macros.

void arcintest(FILE *data_in, FILE *data_out, int size, int arctics[]);

The function arcintest() is the entry point for the arcintest boundary scan operation. The input data is given in the file *data_in*, and the output data is written to the file *data_out*. See Appendix B for details on the formats of these files. The integer *size* is the number of sequences or data patterns to be run.

void load_JTAGDR2_beg(struct data_stream *ds, int arctics[], int data_length);

load_JTAGDR2_beg() loads the JTAG Data Register with *ds* and puts the returned data in *ds*. load_JTAGDR2_beg() inserts a bit into the data stream for each bypassed Arctic before sending the data. It also removes these bypass bits from the output data before returning it in *ds*. This function is intended as the first scan of an arcintest operation. It assumes the TAP controller starts in the RTI state and leaves it in the Pause-DR state. This function will take the TAP controller through the following states:

Run-Test/Idle	(assumes this starting state)
Select-DR-Scan	
Capture-DR	
Shift-DR	(will likely spend multiple cycles in this state)
Exit1-DR	
Pause-DR	

void load_JTAGDR2(struct data_stream *ds, int arctics[], int data_length);

load_JTAGDR2() is identical to load_JTAGDR2_beg() described above except it assumes that the TAP controller starts in the Pause-DR state. This function is intended for all scans of arcintest after the first one. This function will take the TAP controller through the following states:

Pause-DR	(assumes this starting state)
Exit2-DR	
Update-DR	
Run-Test/Idle	
Select-DR-Scan	
Capture-DR	
Shift-DR	(will likely spend multiple cycles in this state)
Exit1-DR	
Pause-DR	

void load_JTAGDR2_end();

load_JTAGDR2_end() simply returns the TAP controller to the Run-Test/Idle state. It is used to end the arcintest operation and assumes that the TAP controller

starts in the Pause-DR state. This function will take the TAP controller through the following states:

Pause-DR	(assumes this starting state)
Exit2-DR	
Update-DR	
Run-Test/Idle	

3.2.3 Implementation Details

The input and output data for the arcintest operation are stored in files. Both input and output files have specific formats. ANCIS expects a particular format when reading the input file and writes output data in a particular format. See Appendix B on page 60 for the input and output file formats. For more detailed information on the boundary scan cells see Appendix A.

The arcintest() function directly calls special JTAG transaction macros, bypassing some of the layers of abstraction. The special JTAG transaction macros are required because the TAP controller needs to be left in the Pause-DR state between data patterns. Other JTAG transaction macros leave the TAP controller in the Run-Test/Idle state. When the JTAG Interface Instruction Register contains the arcintest instruction and the TAP controller is in the Run-Test/Idle state, a system clock is issued for every test clock. If the TAP controller were left in the Run-Test/Idle state, the system clock would continue to cycle, between data patterns. Instead, the TAP controller is left in the Pause-DR state and only spends one cycle in Run-Test/Idle for each data pattern. The result is that one system clock is issued for each complete load of the boundary scan cells.

When data is first scanned into the boundary scan cells for an arcintest operation, the original values of the boundary scan cells are simultaneously scanned out. These original values are saved until the end of the arcintest operation. They are then scanned into the boundary scan cells on the last scan. As the last stream of output data is retrieved, the original scan cell values are replaced. When this is done in practice, the replaced data seems to

be offset by one bit from the original data. The cause of this discrepancy has not yet been determined. One possibility is that the boundary scan cell ring is actually one bit shorter than the documentation indicates.

3.3 Board Control

3.3.1 Functions

The functions used for interfacing to the Corelis JTAG controller board are heavily based on the test software provided by Corelis. These functions also use lower level utility functions from the Corelis software. For documentation on the utility functions and other functions provided by Corelis including the self-test procedures, see Appendix C on page 64. For details on all functions used by ANCIS for hardware control, refer to the source code file "board2.c" in Appendix D.18 on page 165.

int board_reset(void);

This function performs a hard reset and initialization of the JTAG controller. It initializes the base address, sets the TCK frequency, and enables RS-422.

void scan_ir(unsigned short test_bus, unsigned short *output, unsigned short length, unsigned short *input);

This routine scans an arbitrary length bit stream into a target TAP instruction register. scan_ir() gets the data to be output to the JTAG board from address *output* and stores the returned data at address *input*. *length* specifies the number of bits to be scanned.

void scan_dr(unsigned short test_bus, unsigned short *output, unsigned short length, unsigned short *input);

This function scans an arbitrary length bit stream into a target TAP data register. scan_dr() gets the data to be output to the JTAG board from address *output* and stores the returned data at address *input*. *length* specifies the number of bits to be scanned.

void scan_dr2(unsigned short test_bus, unsigned short *output, unsigned short length, unsigned short *input, unsigned short begin);

This function is based on scan_dr() described above. The only difference is in the TAP states that are entered and the extra parameter, *begin*. This variable is a flag indicating whether or not this is the first time the function is entered during one test. If *begin* is equal to BEG, this is the beginning of the test, and the TAP control-

ler should be in the RTI state. Otherwise the TAP controller should be in the Pause-DR state. In either case, the TAP controller is left in the Pause-DR state when the function returns. scan_dr2() gets the data to be output to the JTAG board from address *output* and stores the returned data at address *input*. *length* specifies the number of bits to be scanned.

void scan_dr2_end(unsigned short test_bus);

This function moves the TAP controller from the Pause-DR state to the Run-Test/Idle (RTI) state. This is used after scan_dr2() has been used leaving the TAP controller in the Pause-DR state.

3.3.2 Implementation Details

The process of scanning data into the JTAG interface is essentially the same for all of the above functions. First the TAP controller is moved to the Shift-DR or Shift-IR state by writing a value to the TMS register on the controller board. When the TAP controller reaches the correct state, TCK must stop until the TDO data is ready to be sent. This is done by setting a bit in the MODE0 register of the parallel/serial converter. This bit causes a TCK dependency on the TMS buffer. When the buffer is empty, TCK will stop. When the data is ready, a TCK dependency on TDO and TDI is established so that TCK will cycle only when data is ready. Once all the data has been sent, a TCK dependency on TMS is restored, and a value written to the TMS buffer transitions the TAP controller state. This final state transition starts from the Exit1-DR or Exit1-IR state, because upon completing the data transmission, the parallel/serial converter sends a high bit on the TMS line, moving the TAP controller out of the Shift-DR or Shift-IR state.

An alternate process of scanning data into the JTAG interface utilizes the board's FIFOs. This process involves setting the turbo mode on the board and writing data to the FIFOs rather than directly to the registers on the parallel/serial converter. Writing directly to the register only allows writing one byte at a time. Writes to the FIFO are in word increments. The functions scan_dr_turbo() and scan_ir_turbo() use this process and can replace scan_dr() and scan_ir() respectively. Currently the turbo mode has not been used.

Chapter 4

Testing

4.1 Testing Procedure

Just as important as the details of implementation are the procedures used for testing. These testing practices are relevant to continued development and testing as well as to the evaluation of the current status of ANCIS. This section describes the methods that have been used to test ANCIS.

4.1.1 Verilog Testing Environment

Thus far all of the testing of ANCIS has been done using a Verilog model of an Arctic. This is a software model which simulates the Arctic circuitry. The Verilog model only simulates one Arctic, so it is difficult to simulate the entire Arctic Switch Fabric Network. As a result it has been very difficult to test the functionality of ANCIS on multiple maintenance interfaces arranged in a ring (Figure 2.2).

ANCIS interacts with the Verilog model through the use of three files which are accessible to both ANCIS and the Verilog simulation. These files simulate FIFOs for the TMS, TDO, and TDI signals. The functions which access these files are in the file "sim.c" (Appendix D.20 on page 223). These functions are responsible for the interaction with the Verilog model. Testing with the Verilog model, as opposed to normal operation with the Corelis board, requires alternate versions of the files "board.c" and "jtag_trans.c" as well as "sim.c". This is due to the fact that the versions of "board.c" and "jtag_trans.c" for use with the Corelis board contain functions that are specific to the hardware.

4.1.2 Testing Strategy

The strategy for testing ANCIS has been to test each of the Arctic operations individually using the script interface and verify their operation as much as possible within the testing constraints. Refer to section 3.1.1 for details on using the script interface. Tests

consist of writing test data to writable registers and reading the values of all data registers. The results of these tests are checked for consistency with other tests as well as with the known values of the registers in Verilog.

Testing with the Verilog model is limited in some ways. Since I am not very familiar with Verilog, I have not been able to modify the test at all from the Verilog side. This is limiting because I cannot force events in Arctic which could then be accessed through the maintenance interface. Forcing specific situations in the model would be necessary for testing boundary cases of ANCIS operation. Testing the read of the error counter, for instance, requires the ability to force errors which could then be counted. Another limitation of the tests that have been run thus far is that detailed knowledge of Arctic in terms of register values has not been involved in constructing the tests. This will be necessary at some point, as fairly precise tests will be needed for final validation of ANCIS.

4.2 Usage Details

A test of ANCIS is run by starting two separate processes. One is the execution of ANCIS, and the other is the execution of a test program on the Verilog model. First ANCIS should be started. The code for ANCIS is currently in the directory /home/jj/cwward/ANCIS/code/, and the Makefile calls the ANCIS executable "test". This executable takes the name of the input script file as command line argument.

```
test <script_file_name>
```

Once ANCIS has begun reading the script file and executing script commands, it will stop to wait for Verilog to run. The Verilog test is run from the directory /home/prj4/arcticdf/arctic20testelth/run_test/. The executable is called "run_test" and takes one command line argument, the test file "elthtest".

```
run_test elthtest
```

There are several additional files that are relevant to the Verilog test. The file "/home/prj4/arcticdf/arctic20testelth/maint/maint_node.v" contains Verilog code for the interface with ANCIS. The file "/home/prj4/arcticdf/arctic20testelth/sequences/test003/test003.code" is the test sequence called by the file "elthtest". Finally the file "/home/prj4/arcticdf/arctic20testelth/test_drv/test_template.v" has a section that controls the information included in the Verilog status window.

After Verilog is started and has compiled, it will begin interacting with ANCIS. The progression of the test can be monitored using the Verilog status window, ANCIS debug messages, or an ANCIS log file. The Verilog status window starts automatically when the Verilog test is executed. It displays information about the values of registers in the Verilog model of Arctic. This window is useful for quickly checking that Arctic registers are being read and written correctly.

The ANCIS debug messages and log file entries are status messages printed from many points throughout the ANCIS code. Debug messages are printed to standard output. These messages are printed based on the value of the constant *TEST* in ancis.h (Appendix D.1 on page 71). A value of zero disables the debug messages completely. Values of 1-7 specify various degrees of detail. Seven causes all debug messages to be printed; one requests only the highest level messages including any error messages. Since the source code files represent the levels of abstraction of the system, each debug level corresponds to the messages originating in one source file. Log file entries are the same as the debug messages except that they depend on the global variable *logf_level.* This variable is set to zero, suppressing log entries, unless the start_log command is given in the script file. With this command the level can be set, again from 1-7 with the same file correspondence as the debug messages.

4.3 Preliminary Results

This section details several test scripts and the results of the tests run with these scripts. Each of these test scripts targets several related ANCIS commands. All of these tests illustrate the use of log files, and several also test some aspect of the reset command.

4.3.1 Control Register

This first test file is designed to test the ANCIS commands that access the port/statistics control register. These commands are **read_psc** and **write_psc**.

```
start_log test_psc.log

reset_controller

reset
ARCTIC 0

read_psc
ARCTIC 0

write_psc
ARCTIC 0
13717

read_psc
ARCTIC 0

end_log
```

The **reset** command causes a value of hexadecimal 13333 to be loaded into the control register. This is confirmed by the "control reg" value in the Verilog status window and again by the result of the first **read_psc** command. The next command is **write_psc** with a value of hexadecimal 13717. This number appears in the status window and is also successfully read back with the subsequent **read_psc** command.

```
Begin log file test_psc.log at level 1.

reset_controller

reset
```

```
arctics = 1

read_psc
psc read = 010011001100110011

write_psc
data =
010011011100010111
arctics = 1

read_psc
psc read = 010011011100010111
```

4.3.2 Configuration and Mode Registers

The ANCIS commands that access the configuration and mode registers are the targets of the following test script. These commands are **change_mode**, which writes the mode register, **cfg_read_port**, **cfg_write_port**, and **cfg_write_all**, which access the configuration registers.

```
start_log test_cfg.log

reset_controller

reset
ARCTIC 0

#reset leaves Arctic in configuration mode

cfg_write_all
reg1
ARCTIC 0
020c4100
reg2
ARCTIC 0
08041cc5
reg3
ARCTIC 0
f0001000
reg4
ARCTIC 0
f0002000
reg5
```

```
ARCTIC 0
000300ff

#read cfg registers on port 3
cfg_read_port
3
ARCTIC 0

cfg_write_port
3
reg1
ARCTIC 0
0ffbdff8
reg2
ARCTIC 0
040f7bfe
reg3
ARCTIC 0
00000000
reg4
ARCTIC 0
00000000
reg5
ARCTIC 0
000300ff

change_mode
normal
ARCTIC 0

change_mode
config
ARCTIC 0

cfg_read_port
3
ARCTIC 0

end_log
```

This test shows that the **reset** command successfully leaves Arctic in configuration mode. It also successfully changes the mode from configuration to normal and back to configuration. This test first writes the configuration registers on all ports using the

cfg_write_all command. This write is verified by the Verilog status window as well as the subsequent read of the port 3 configuration registers. Following this read, the port 3 configuration registers are written with new values. Again this write is confirmed with both the Verilog status window and a subsequent read of the port 3 configuration registers.

```
Begin log file test_cfg.log at level 1.

reset_controller

reset
arctics = 1

cfg_write_all
data =
00000000100000110001000001000000000
01000010000000010000011100110001010
00111100000000000000010000000000000
00111100000000000000010000000000000
11000000000000001100000000111111111
arctics = 1

cfg_read_port
port 3
arctics = 1

cfg_read_port =
00000000100000110001000001000000000
00000010000000010000011100110001010
00111100000000000000010000000000000
00111100000000000000010000000000000
00000000000000001100000000111111111

cfg_write_port
port 3
data =
00000011111111101111011111111111000
10000001000000111101111011111111110
00000000000000000000000000000000000
00000000000000000000000000000000000
11000000000000001100000000111111111
arctics = 1

change_mode
normal
```

```
arctics = 1

change_mode
config
arctics = 1

cfg_read_port
port 3
arctics = 1

cfg_read_port =
00000011111111101111011111111111000
00000001000000111101111011111111110
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000001100000000111111111
```

4.3.3 Error Counters

This script tests the ANCIS command **errors** which reads the Arctic error counters. The command is used twice. The first reads and clears the error counters. The second is a nondestructive read.

```
start_log test_error.log

reset_controller

reset
ARCTIC 0

errors
Clear
ARCTIC 0

errors
No
ARCTIC 0

end_log
```

Both reads of the error counters indicate zero errors, and this is confirmed by the Verilog status window.

```
Begin log file test_error.log at level 1.

reset_controller

reset
arctics = 1

errors
clear 1
arctics = 1

errors =
00000000000000000000000000000000000000000000000000
0000000000

errors
clear 0
arctics = 1

errors =
00000000000000000000000000000000000000000000000000
0000000000
```

4.3.4 Buffer Free Counters

This test is for the command **bufs_free** which reads the buffer free counters. The configuration registers need to be written prior to reading the buffer free counters so that the values to be read are defined.

```
start_log test_buf.log

reset_controller

reset
ARCTIC 0

cfg_write_all
reg1
ARCTIC 0
020c4100
reg2
ARCTIC 0
08041cc5
reg3
```

```
ARCTIC 0
f0001000
reg4
ARCTIC 0
f0002000
reg5
ARCTIC 0
000300ff

bufs_free
Arctic 0

end_log
```

Although the Verilog status window indicates that the buffer free register for each port holds the value 3, the read of the counters using the **bufs_free** command returns a different value. The value read by ANCIS from the Verilog model is that of undefined data. The data stream value does not indicate a true value, only that the data was undefined. These results seem to be associated with some side effects of a reset operation in the test system. This appears to be an issue only within the Verilog testing environment.

```
Begin log file test_buf.log at level 1.

reset_controller

reset
arctics = 1

cfg_write_all
data =
00000000100000110001000001000000000
01000010000000010000011100110001010
00111100000000000000001000000000000
00111100000000000000010000000000000
11000000000000001100000000111111111
arctics = 1

bufs_free
arctics = 1

bufs_free = 01100010011000100010
```

4.3.5 Statistics Registers

The commands that access the statistics registers are **clear_stats**, **port_stat**, **port_stats**, and **all_stats**. These commands will be tested with a script like the following. Notice that the control register must be written with the hexadecimal value 0x30000 in order to enable the input ports before reading the statistics registers.

```
reset_controller

reset
ARCTIC 0

change_mode
normal
ARCTIC 0

write_psc
ARCTIC 0
30000

port_stat
0 PACKETS Clear
ARCTIC 0

port_stats
0
ARCTIC 0

clear_stats
ARCTIC 0

all_stats
no-clear
ARCTIC 0
```

Currently these ANCIS commands are not able to clear any of the statistics registers, so any subsequent read returns only undefined values. This is thought to be associated with a side effect of the reset operation in the test system. The problem seems to be unique to the Verilog test system and is not expected to appear when ANCIS is used with the Corelis JTAG controller board.

4.3.6 Arcintest

For this test of the **arcintest** command, a value is written to the control register before arcintest is performed, and the control register is read after arcintest is done. This is used to check that the original boundary scan cell data is returned after arcintest.

```
start_log test_arc.log

reset_controller

reset
ARCTIC 0

change_mode
normal
ARCTIC 0

write_psc
ARCTIC 0
21117

arcintest
in.dat out.dat
ARCTIC 0

read_psc
ARCTIC 0

end_log
```

Two patterns were scanned for this test. The following is the input data file.

```
2
3f ff f
1 1 ff ff ff ff
1 1 ff ff ff ff
1 1 ff ff ff ff
1 1 ff ff ff ff
00 00 0
0 0 00 00 00 00
0 0 00 00 00 00
0 0 00 00 00 00
0 0 00 00 00 00
```

The following log file shows that the replaced data seems to be offset by one bit from the original data. This problem may be associated with the size of the boundary scan cell ring.

```
Begin log file test_arc.log at level 1.

reset_controller

reset
arctics = 1

change_mode
normal
arctics = 1

write_psc
data =
100001000100010111
arctics = 1

arcintest
input file: in.dat, output file: out.dat
2 patterns
arctics = 1

read_psc
psc read = 010000100010001011
```

The output data file seems to indicate the same one bit shift in the port/statistics control register. The rest of the output data is difficult to interpret without more detailed information about the activity in the Arctic Verilog model.

```
1f ff f 1
55 55 aa aa 1 0 1 0
55 55 aa aa 1 0 1 0
55 55 aa aa 1 0 1 0
55 55 aa aa 1 0 1 0
end pattern 1

 0  0 0 0
55 55 aa aa 1 0 0 1
55 55 aa aa 1 0 0 1
55 55 aa aa 1 0 0 1
```

```
55 55 aa aa 1 0 0 1
end pattern 2
```

4.4 Future Testing

Much testing of ANCIS remains to be done as the application of the system progresses toward its goal. The next step in testing will be to test ANCIS using the Corelis board. Once the Corelis board is in use and Arctics are available, testing the system on multiple Arctics will finally be possible. In addition to these, future tests will need to incorporate a detailed knowledge of Arctic in terms of data values. These are the primary areas of the system that have not yet been tested and will be priorities.

At this time, ANCIS has not yet been tested with the Corelis board. This is an extremely high priority since ANCIS cannot be used with a physical Arctic until the system is working with the Corelis board. The first step will be to use the loop back facility on the board to test ANCIS without using an Arctic. This type of testing should be useful for testing the basic functionality of the software drivers and the board interface. The next step will be to test with an Arctic to more thoroughly verify the board and the entire system.

Since the Verilog model only simulates one Arctic, it has not been possible to test ANCIS on more than one Arctic using the current Verilog configuration. The step from one Arctic to multiple chips in a scan chain is not a trivial one. This step may require some time and effort in debugging. Testing this aspect of ANCIS will only be possible when the Verilog test has been modified or when the system is functioning with the Corelis board and Arctics are available for test use. This will likely be one of the later testing phases.

One aspect of ANCIS testing that can be improved at any time is the care that is put into choosing data for test cases. Up to this point in the testing process, only the basic

operation of ANCIS operations has been verified. To this end only dummy data, meaningless to Arctic, has been needed. More precise testing will require meaningful data for writes to configuration registers, the control register, and the boundary scan cells for arcintest. Only then will tests be able to cover boundary cases and possible trouble spots. This will allow a much greater degree of confidence in the operation of ANCIS for all cases.

Chapter 5

Conclusions

ANCIS is on the verge of becoming very useful for the testing and maintenance of the Arctic Switch Fabric Network. ANCIS development has been taken nearly as far as it can go in the Verilog testing environment. The system is ready to be developed with the JTAG controller board and will then be able to test chips. Some further work will be needed to reach this goal. In addition future projects will be needed in order to reach the long term goals of the ANCIS project.

ANCIS is currently capable of performing many functions within the Verilog testing environment. The script interface has reached a functional level. The interface is robust and allows some flexibility, although not every possible feature has been implemented. With a Verilog model of Arctic, ANCIS can successfully reset the maintenance interface, read and write the port/statistics control register and the configuration registers, change the mode of operation, execute an arcintest boundary scan operation, and read the error counters. Functions which access the statistics and buffer free registers have not yet been successful, but the problems seem to be specific to the Verilog testing environment.

The next step in development is to test ANCIS with the Corelis JTAG controller board. The necessary software, including ANCIS modifications and hardware drivers, has been completed and is in place for use with the board. This software does however need to be tested with the hardware. This will likely be an immediate extension of the current system. Once ANCIS is operational with the JTAG controller board, the arcintest boundary scan operation can be used to test physical Arctics. Getting to this stage is a high priority and will probably be a straightforward extension of the current system.

More work will be required in order to reach longer term goals. To make ANCIS a more complete system in later versions, several additions could be made. Eventually the extest boundary scan function may be desired. This function would allow testing externally of the chip. The low level scan function may also be useful at some point. This function allows for increased internal Arctic testing. In the long run, it may also be worth the effort to add some extra features to the script interface. These features will probably depend on the way in which the script interface is most commonly being used.

Much future work may be aimed at interfacing ANCIS to the StarT-Voyager for run-time control of the Arctic Switch Fabric Network. This is the long term goal for the ANCIS project. The system would then be able to access data from the network for analysis and error detection. The StarT-Voyager would be able to configure the network through ANCIS, making run-time changes. In this role, ANCIS would be critical to the maintenance of the Arctic Switch Fabric Network.

References

[1] Boughton, Andy, et al., DRAFT Arctic User's Manual, August 1995.

[2] Corelis Inc., PC-1149.1/100F High Speed PC-AT Bus Boundary-Scan Controller Board User's Manual, 1993-1994.

[3] Ogston, Elizabeth F. Y., The Arctic Switch Fabric Control Interface System, February 1996.

Appendix A

Boundary Scan Ring Cells

This file ... /home/prj4/arcticdf/arctic22df/control_test/mrings/bsc.def

```
//note: frame_xt and bf_xt are "sample only" flops for extest
//note: oclk_q and oclk_nq are "stimulus only" flops for extest

//added error pin FF on 1/25/94

wire [322:0] bsc_ring;                  //was [641:0]
assign bsc_ring = {
 u1.control_bus[17:0],                  //undefined for extest; stimulus for intest.

//error pin 1-bit
 u1.ip_error_q,                         //stimulus for extest; sample for intest

 u1.ip3.u9.iclk_q,                      //sample for extest; don't care for intest.
 u1.ip3.u9.iclk_nq,                     //sample for extest; don't care for intest.
 u1.ip3.u9.frame_xt,                    //sample for extest; don't care for intest.
 u1.ip3.u9.frame_ipx,                   //undefined for extest; stimulus for intest.
 u1.ip3.u9.bf_xt,                       //sample for extest; don't care for intest.
 u1.ip3.u9.bf2,                         //undefined for extest; stimulus for intest.

 u1.ip3.u9.data_ipx[31:0],              //data_ipx[15:0]
                                        //undefined for extest; stimulus for intest.
                                        //data_ipx[31:16]
                                        //sample for extest; stimulus for intest.
 u1.ip2.u9.iclk_q,
 u1.ip2.u9.iclk_nq,
 u1.ip2.u9.frame_xt,
 u1.ip2.u9.frame_ipx,
 u1.ip2.u9.bf_xt,
 u1.ip2.u9.bf2,
 u1.ip2.u9.data_ipx[31:0],

 u1.ip1.u9.iclk_q,
 u1.ip1.u9.iclk_nq,
 u1.ip1.u9.frame_xt,
```

```
u1.ip1.u9.frame_ipx,
u1.ip1.u9.bf_xt,
u1.ip1.u9.bf2,
u1.ip1.u9.data_ipx[31:0],

u1.ip0.u9.iclk_q,
u1.ip0.u9.iclk_nq,
u1.ip0.u9.frame_xt,
u1.ip0.u9.frame_ipx,
u1.ip0.u9.bf_xt,
u1.ip0.u9.bf2,
u1.ip0.u9.data_ipx[31:0],

u1.op0.outpad_0.data_shad[31:0],     //data_shad[15:0]
                                     //don't care for extest; sample for intest
                                     //data_shad[31:16]
                                     //stimulus for extest; sample for intest

u1.op0.outpad_0.frame_reg1,          //stimulus for extest; sample for intest.
u1.op0.outpad_0.frame_reg,           //stimulus for extest; sample for intest.
u1.op0.outpad_0.buff_free_reg1,      //stimulus for extest; sample for intest.
u1.op0.outpad_0.buff_free_reg,       //stimulus for extest; sample for intest.
u1.op0.outpad_0.oclk_q,              //stimulus for extest; don't care for intest.
u1.op0.outpad_0.oclk_nq,             //stimulus for extest; don't care for intest.

u1.op1.outpad_0.data_shad[31:0],
u1.op1.outpad_0.frame_reg1,
u1.op1.outpad_0.frame_reg,
u1.op1.outpad_0.buff_free_reg1,
u1.op1.outpad_0.buff_free_reg,
u1.op1.outpad_0.oclk_q,
u1.op1.outpad_0.oclk_nq,

u1.op2.outpad_0.data_shad[31:0],
u1.op2.outpad_0.frame_reg1,
u1.op2.outpad_0.frame_reg,
u1.op2.outpad_0.buff_free_reg1,
u1.op2.outpad_0.buff_free_reg,
u1.op2.outpad_0.oclk_q,
u1.op2.outpad_0.oclk_nq,
```

```
u1.op3.outpad_0.data_shad[31:0],
u1.op3.outpad_0.frame_reg1,
u1.op3.outpad_0.frame_reg,
u1.op3.outpad_0.buff_free_reg1,
u1.op3.outpad_0.buff_free_reg,
u1.op3.outpad_0.oclk_q,
u1.op3.outpad_0.oclk_nq
};
```

Appendix B

Arcintest File Formats

Boundary scan cell info can be found in the file
/home/prj4/arcticdf/arctic22df/control_test/mrings/bsc.def
The usage of the function arcintest is based on this information.

The boundary scan ring is 323 bits.
The first 18 bits are the psc_reg
The next bit is the error pin u1.ip_error_q
Then 4 input ports of 38 bits each
and finally 4 output ports of 38 bits each

The input file will contain data for cells marked as "stimulus for intest" in the file bsc.def and the output file will contain data for cells marked as "sample for intest". The output file will contain the psc register in addition.

INPUT FILE
The format of the input file for arcintest should be as follows:
(All numbers will be interpreted as hexadecimal.)

Separate bytes (two hex digits) with a space: ff ff a5 ec
Line 1: The number of patterns (# of sets of data for the scan ring)
(THIS IS IN HEX ALSO - to be consistent)
[Pattern 1]
Line 2: 16 bit psc [padded with zeros on the left]
Line 3: Input Port 3 - 1 bit frame_ipx, space, 1 bit bf2, space, 32 bit data_ipx
Line 4: Input Port 2 - "
Line 5: Input Port 1 - "
Line 6: Input Port 0 - "
[Pattern 2]
Line 7: 16 bit psc
Line 8: Input Port 3 - 1 bit frame_ipx, space, 1 bit bf2, space, 32 bit data_ipx
. . .

Here is an example: There are two patterns (2 on the first line).
Every input bit is 1 for both patterns.

2
3f ff f
1 1 ff ff ff ff
1 1 ff ff ff ff
1 1 ff ff ff ff
1 1 ff ff ff ff
3f ff f
1 1 ff ff ff ff
1 1 ff ff ff ff
1 1 ff ff ff ff
1 1 ff ff ff ff

OUTPUT FILE
The format of the output file for arcintest will be as follows:
(All numbers will be interpreted as hexidecimal.)

[Pattern 1]
Line 1: 16 bit psc, space, 1 bit error_q
Line 2: Output Port 0 - 32 bit data_shad, space, each of the 1 bit values
frame_reg1, frame_reg, buff_free_reg1, and
buff_free_reg separated by spaces.
Line 3: Output Port 1 - "
Line 4: Output Port 2 - "
Line 5: Output Port 3 - "
Line 6: "end pattern 1"
[Pattern 2]
Line 7: 16 bit psc, space, 1 bit error_q
Line 8: Output Port 0 - 32 bit data_shad, ...
. . .

Here is an example: There are two patterns.
Every output bit is 1 for both patterns.

3f ff f 1
ff ff ff ff 1 1 1 1
ff ff ff ff 1 1 1 1
ff ff ff ff 1 1 1 1
ff ff ff ff 1 1 1 1
end pattern 1

3f ff f 1
ff ff ff ff 1 1 1 1
ff ff ff ff 1 1 1 1
ff ff ff ff 1 1 1 1
ff ff ff ff 1 1 1 1
end pattern 2

Appendix C

Corelis Software Documentation

Documentation from the file test.c

PC-1149.1/100F Boundary-Scan Controller Self-Test Software

Copyright (c) 1994, Corelis Inc.

Software Version: 1.1 [August 21, 1994]

Modification History:
21-Aug-94 SGH TX FIFO size changed to 1Kx8. Updated FIFO flags test and TX FIFO size constant.

Usage:
TEST

This program provides sample control code for the Corelis PC-1149.1/100F Boundary-Scan Controller, which is based on the National Semiconductor SCANPSC100F Boundary-Scan Parallel/Serial Converter.

The functions provided in this listing are useful in writing general purpose control code for the PC-1149.1/100F. The main program provided gives an example of use of the functions to provide board self-test.

This program was compiled with Borland C++ v3.1.

Notes: The PC1149.1/100F has 2 SCANPSC100F parallel-to-serial converters, providing a total of 4 JTAG ports. For purposes of documentation, the first parallel-to-serial converter is referred to as PSC1 (for JTAG busses 1 and 2) and the second parallel-to-serial converter is referred to as PSC2 (for JTAG busses 3 and 4).

The functions provided in this package can be separated into four categories: test functions, high-level access functions, low-level access functions and utility functions. A list of the functions

(in the order of occurrance in the file) is provided below:

wait_for_ready()
check_ready()
write_psc()
read_psc()
write_tdo()
write_tms()
read_tdi()
load_cnt_32()
read_cnt_32()
get_byte_word()
build_word()
pgrm_scan_clk()
soft_reset()
hard_reset()
wait_keypress()
setbit()
getbit()
enable_jtag_selftest()
disable_jtag_selftest()
center()
scan_ir()
scan_dr()
circulate_dr()
scan_ir_turbo()
scan_dr_turbo()
reg_reset_test()
psc_loopback()
test_cnt_32()
output_rdbk_test()
jtag_selftest()
jtag_selftest_turbo()
fifo_flags_test()
p_to_s_conv_test()

The main procedure (main) is the last procedure in the file. The functions are listed below in their respective categories:

Test Functions

These functions perform tests of one or more parts of the PC-1149.1/100F.

fifo_flags_test() - performs a test of the FIFO flags
jtag_selftest() - performs a self-test of all JTAG channels (without using turbo mode)
jtag_selftest_turbo() - performs a self-test of all JTAG channels (using turbo mode)
output_rdbk_test() - performs a test of the parallel I/O
psc_loopback() - performs an internal loopback test of one of the JTAG signals on a parallel/serial converter
p_to_s_conv_test() - calls lower-level tests to test both parallel/serial converters
reg_reset_test() - checks registers in a parallel/serial converter after reset
test_cnt_32() - tests the 32-bit counter in a parallel/ serial converter

High-Level Access Functions

These functions are used to setup or perform scanning operations with the PC-1149.1/100F. These are the functions which will be used when perfoming most operations with the PC-1149.1/100F.

circulate_dr() - used to circulate data through target data register.
disable_jtag_selftest() - disables self-test of the on-board SCAN18374
enable_jtag_selftest() - enables self-test of the on-board SCAN18374 from one of the four JTAG channels
hard_reset() - performs a reset of the FIFOs, parallel/ serial converters and target
pgrm_scan_clk() - used to select the clock source and frequency for JTAG channels 1 and 2 or JTAG channels 3 and 4
scan_dr() - used to scan data into a target data register. This function uses the TDO

and TDI buffers on parallel/serial converter.

scan_dr_turbo() - used to scan data into a target data register. This function uses the turbo parallel/serial converters

reg_reset_test() - checks registers in a parallel/serial converter after reset

test_cnt_32() - tests the 32-bit counter in a parallel/serial converter

High-Level Access Functions

These functions are used to setup or perform scanning operations with the PC-1149.1/100F. These are the functions which will be used when perfoming most operations with the PC-1149.1/100F.

circulate_dr() - used to circulate data through target data register.

disable_jtag_selftest() - disables self-test of the on-board SCAN18374

enable_jtag_selftest() - enables self-test of the on-board SCAN18374 from one of the four JTAG channels

hard_reset() - performs a reset of the FIFOs, parallel/serial converters and target

pgrm_scan_clk() - used to select the clock source and frequency for JTAG channels 1 and 2 or JTAG channels 3 and 4

scan_dr() - used to scan data into a target data register. This function uses the TDO and TDI buffers on parallel/serial converter.

scan_dr_turbo() - used to scan data into a target data register. This function uses the turbo mode of operation. Data is written to and read from on-board FIFOs instead of using TDO and TDI buffers on parallel/serial converters.

scan_ir() - used to scan an instruction into the

target instruction register. This function uses the TDO and TDI buffers on the parallel/serial converter.

scan_ir_turbo() - used to scan an instruction into the target instruction register. This function uses the turbo mode of operation. Data is written to and read from on-board FIFOs instead of using TDO and TDI buffers on parallel/serial converter.

soft_reset() - performs an internal reset of one of the parallel/serial converters.

Low-Level Access Functions

These functions are used to perform low-level operations on the PC-1149.1/100F.

check_ready() - used to determine if parallel/serial converters are ready to be accessed

load_cnt_32() - used to load the 32-bit counter on a parallel/serial converter

read_cnt_32() - used to read the 32-bit counter on a parallel/serial converter

read_psc() - used to read a register on a parallel/serial converter

read_tdi() - used to read the TDI buffer on a parallel/serial converter

wait_for_ready() - used to determine when parallel/serial converter is ready to be accessed.

write_psc() - used to write data to a register on a parallel/serial converter

write_tdo() - used to write data to the TDO buffer on a parallel/serial converter

write_tms() - used to write data to a TMS buffer on a parallel/serial converter

Utility Functions

These are general purpose routines used throughout the self-test program.

build_word()	- combines two bytes into a word
center()	- centers and displays a string on the screen
getbit()	- used to get the value of a bit in an arbitrary length array.
get_byte_word()	- used to get MSB or LSB in a word
setbit()	- used to set a bit in an arbitrary length array.
wait_keypress()	- waits for RETURN or ESC to be pressed

Appendix D

ANCIS Source Code

This section contains the C language source code for ANCIS. Currently this code can be found in the directory /home/jj/cwward/ANCIS/code/ along with all other relevant files.

Several pairs of files are meant as alternates depending on the test or operation environment. Testing with Verilog requires the use of the files "jtag_trans1.c", "board1.h", board1.c", "sim.h", and "sim.c". In that case the following files should not be used: "jtag_trans2.c", "board2.h", and board2.c". When ANCIS is used with the Corelis JTAG controller board, the opposite is true. The second set of files should then be used in place of the first set of files.

D.1 ancis.h

```
/*header file for all parts of the ANCIS program and scripts*/

#include <stdio.h>
#include <stdlib.h>

/****** TEST is the debugging level: *******/
/******       0 for none, 7 for full *******/
#define TEST 7

#define NA 1       /*number of arctics in the scan chain*/

#define sparc

#include "data_stream.h"

#include "sim.h"
#include "board.h"
#include "jtag_trans.h"
#include "access_macs.h"
#include "arctic_access_ops.h"
#include "upper_level.h"
```

D.2 data_stream.h

```
/*header file for data_stream abstraction
 Elth 11/28/95
*/

#include <math.h>

enum bit{ZERO, ONE};

struct data_stream
{
 int size;
 unsigned short *data;
};

/*data kept in 16 bit chunks, space allocated when the data_stream is
 created
 bits are labled starting at 0*/

struct data_stream *create_data_stream(int size);
/* returns an empty data stream with size space in it */
/*size = number of bits*/

void delete_data_stream(struct data_stream *ds);
/* deletes the given data stream, deallocates memory used */

void set_bit(struct data_stream *ds, long bit_number, enum bit value);
/*sets the given bit to value */

enum bit get_bit(struct data_stream *ds, long bit_number);
/*returns the value of given bit*/

int set_byte(struct data_stream *ds, long start,  short byte);
/*adds given byte to data stream begining at start*/
/*start is the bit number, not the byte number*/

int get_byte(struct data_stream *ds,  long start, short *byte);
/*returns the next byte in the stream.  successive get_next_byte
 commands will return sucessive bytes
 returns 0 if there are no bytes left
 returns number of valid bits if a byte is returned*/
```

```
int get_ds_length(struct data_stream *ds);

unsigned short *ds2array(struct data_stream *ds);

struct data_stream *append_streams(struct data_stream *ds1,
                                   struct data_stream *ds2);
/*returns a new data stream that consists of ds1 followed by ds2
 assumes both streams are full*/


struct data_stream *parse_to_ds(int data1,
                                int size1,
                                int arctics[NA]);
/*creates a data stream with data1 for each specified
  arctic and no data for each bypasses*/

int add_bits(struct data_stream *ds, int num_bits, int start,
             int bits);
/*add num_bits number of bits to ds starting at start (bit number)
  from the number bits*/

void flip_stream(struct data_stream *ds);
/* reverses the order of the bits in ds */

void fprint_stream(struct data_stream *ds);
/* for writing the bits out to the log file */

void print_arctic_data(struct data_stream *ds, int size, int arctics[]);
/* for writing data out to the log file */

short flip_byte(short);

/* for debugging */
void print_stream(struct data_stream *ds);
void bin_print(int digits, int number);
void print_arctic_data(struct data_stream *ds, int size, int arctics[]);
```

D.3 data_stream.c

```
/* File  containing all data stream functions
   data_stream.h contains declarations.

   Elth 11/28/95
*/

/* data streams are abstractions for the bit streams to be
   fed into the scan chain.  At upper levels they are used to
   pass around data to be read into, or read out of the arctic
   registers, at lower levels they contain control bits as well

   data streams can be written and read bit by bit, or byte by
   byte.  their correct length must be defined when they are
   created.  Functions are also provided to get a streams length,
   create and destroy streams, and combine streams.
*/

#include "ancis.h"


extern  FILE *logf;
extern  int logf_level;


/***********
main()
{
 struct data_stream *temp_s, *temp_s2;
 int i, j;
 unsigned short byte;

 temp_s = create_data_stream(32);
 temp_s2 = create_data_stream(20);


 for(i=0; i<32; i = i+8)
   set_byte(temp_s, i, 170);

 for(i=0; i<20; i = i+8)
   set_byte(temp_s2, i, 204);
```

```
 print_stream(temp_s);
 print_stream(temp_s2);

 print_stream(append_streams(temp_s2, temp_s));

}
**********/

void bin_print(int digits, int number)
{
 int i;
 for(i=0; i<digits; i++)
  printf("%d", ((number & (1<< i)) >> i));
 printf("\n");
}

struct data_stream *create_data_stream(int size)
/*creates an empty data_stream of size size.  returns a pointer to it
 empty data streams have 0's in all their data bytes*/
{
 struct data_stream *ds;
 int i;

 if((ds = (struct data_stream *)malloc(sizeof(struct data_stream))) == NULL)
  {
   printf("uh oh, can't get enough memory \n");
   if (logf_level>0)
       fprintf(logf, "uh oh, can't get enough memory \n");
  }

 ds->size = size;
 size = size/16+1;

 if((ds->data = (unsigned short *)calloc(size, sizeof(short))) == NULL)
  {
   printf("uh oh, can't get enough memory \n");
   if (logf_level>0)
       fprintf(logf, "uh oh, can't get enough memory \n");
  }

 for(i=0; i<size; i++)
  ds->data[i] = 0;
```

```
 return(ds);
}

void delete_data_stream(struct data_stream *ds)
/*deletes the given stream, frees up memory*/
{
 free(ds->data);
 free(ds);
}

void set_bit(struct data_stream *ds, long bit_number, enum bit value)
/*sets the given bit address to the given value*/
{
 int i, j;

 if(bit_number >= ds->size)
  {
   printf("*** attempt to set bit outside data stream: %d, %d\n",
          bit_number, ds->size);
   if (logf_level>0)
       fprintf(logf, "*** attempt to set bit outside data stream: %d, %d\n",
          bit_number, ds->size);
   return;
  }

 if (value == ZERO)
  ds->data[bit_number/16] = ds->data[bit_number/16] &
   ~(1 << (bit_number%16));
 else
  ds->data[bit_number/16] = ds->data[bit_number/16] |
   (1 << (bit_number%16));
}



enum bit get_bit(struct data_stream *ds, long bit_number)
  /*returns bit bit_number from data_stream*/
{
 return((ds->data[bit_number/16] & (1 << (bit_number%16))) >>
        (bit_number%16));
}
```

```
int set_byte(struct data_stream *ds, long start, short byte)
{
/*adds given byte to data stream begining at start*/
/*start is a bit number, not a byte number*/
/*if the byte goes over the lenght of the stream,
  truncates the number by filling in to the end of the stream*/
/*returns number of bits set*/

  int word_num;
  int offset;
  int mask;
  short new_byte;

/*if byte outside the data_stream*/
  if (start > ds->size)
    {
     printf("attempt to add byte to full stream: %d, %d\n",
            ds->size, start);
     if (logf_level>0)
          fprintf(logf, "attempt to add byte to full stream: %d, %d\n",
            ds->size, start);
     return(0);
    }

/* find the offset from the begining of a word  and the word
   number withing the stream */

  offset = start % 16;
  word_num = start / 16;

/* if the entire byte falls within one word */

  if(offset <= 8)
    {
     /* clear the correct byte within the word */
     mask = ~(255 << offset);

     ds->data[word_num] = ds->data[word_num] & mask;
     /*splice in the new byte*/
     byte = byte << offset;
     ds->data[word_num] = ds->data[word_num] | byte;
     return(8);
    }
```

```
/*if the byte is spread over two words*/
  else
    {
     /*first word, lsb of byte*/
     mask = ~(255 << offset);
     ds->data[word_num] = ds->data[word_num] & mask;
     new_byte = byte << offset;
     ds->data[word_num] = ds->data[word_num] | new_byte;

     /*second word, msb of byte*/
     /*if next byte is over the end of the data_stream
          stop*/

     /*check this bound!!!*/
     if((word_num+2) *8 > ds->size)
          return(offset);

     mask = ~(255 >> (16-offset));
     ds->data[word_num+1] = ds->data[word_num+1] & mask;
     new_byte = byte >> (16-offset);
     ds->data[word_num+1] = ds->data[word_num+1] | new_byte;
     return(8);
    }
}

int get_byte(struct data_stream *ds, long start, short *byte)
{
/*returns the 8 bits starting at bit number start as the arg byte
  returns number of bits return.  If they byte goes over the end
  of the data_stream then the lsb's are filled in with 0's*/
  int word_num;
  int offset;
  short mask;
  short new_byte;

/*if byte outside the data_stream*/
  if (start > ds->size)
    {
     printf("attempt to get byte not in data_stream: %d, %d\n",
               ds->size, start);
     if (logf_level>0)
          fprintf(logf, "attempt to get byte not in data_stream: %d, %d\n",
```

```
                ds->size, start);
    return(0);
   }

/* find the offset from the begining of a word  and the word
   number withing the stream */

  offset = start % 16;
  word_num = start / 16;

/* if the entire byte falls within one word */

  if(offset <= 8)
    {
     /* mask off the correct byte within the word, and shift
      it over to the correct position*/

     mask = (255 << offset);
     *byte = ((ds->data[word_num] & mask) >> offset);

     return(8);
    }

/*if the byte is spread over two words*/
  else
    {
     /*first word, lsb of byte*/
     mask = (255 << offset);
     *byte = (ds->data[word_num] & mask) >> offset;

     /*second word, msb of byte*/
     /*if next byte is over the end of the data_stream
          stop*/

     /*check this bound!!!*/
     if((word_num+2) *8 > ds->size)
          return(offset);

     mask = (255 >> (16-offset));

     *byte = (*byte |
          ((ds->data[word_num+1] & mask) << (16-offset)));
```

```
    return(8);
   }
}

int get_ds_length(struct data_stream *ds)
/*returns length of data_stream*/
{
 return(ds->size);
}

unsigned short *ds2array(struct data_stream *ds)
{
 return(ds->data);
}

struct data_stream *append_streams(struct data_stream *ds1,
                                   struct data_stream *ds2)
{
 int size;
 int i, j;
 struct data_stream *new_ds;
 int offset;
 unsigned short mask1, mask2;
 int ds_offset;

 /*create a new data stream*/
 size = ds1->size + ds2->size;
 new_ds = create_data_stream(size);

 /*transfer over data from ds1*/

 for(i=0; i<= ds1->size/16; i++)
  new_ds->data[i] = ds1->data[i];

 /*transfer over data from ds2*/

 offset = ds1->size % 16;
 mask1 = 65535 >> offset;
 mask2 = 65535 << (16-offset);
 ds_offset = ds1->size/16;

 /*clear overflow bits from ds1*/
 new_ds->data[ds_offset] = new_ds->data[ds_offset] & ~(65535 << offset);
```

```
 /*for each word in ds2, split it in half at offset, and
   add two halfs one at a time*/

 for(i=0; i<= ds2-> size /16; i++)
   {
    /*first half of ds2 word*/
    new_ds->data[ds_offset+i] = new_ds->data[ds_offset+i] |
        ((ds2->data[i] & mask1) << offset);

    /*second half of ds2 word*/
    new_ds->data[ds_offset+i+1] =
    ((ds2->data[i] & mask2) >> (16 - offset));
   }

 return(new_ds);
}

void flip_stream(struct data_stream *ds)
{
/* reverses order of bits in the data stream */

 int i;
 enum bit temp;

 for (i=0; i < (ds->size)/2; i++)
   {
    temp = get_bit(ds, ds->size - 1 - i);
    set_bit(ds, ds->size - 1 - i, get_bit(ds, i));
    set_bit(ds, i, temp);
   }
}

short flip_byte(short byte)
{
/* reverses the order of bits in the byte */

 return(((byte&1)<<7) |
        ((byte&2)<<5) |
        ((byte&4)<<3) |
        ((byte&8)<<1) |
        ((byte&16)>>1) |
        ((byte&32)>>3) |
```

```
        ((byte&64)>>5) |
        ((byte&128)>>7) );
}

void fprint_stream(struct data_stream *ds)
{
 /*print each bit of stream, in binary, to the log file*/
 /*msb, the last bit in the stream now comes first*/
 int i;

 for(i=(ds->size - 1); i>=0; i--)
  fprintf(logf, "%d", get_bit(ds, i));

 fprintf(logf, "\n");
}

void print_stream(struct data_stream *ds)
{
 /*print each bit of stream, in binary*/
 /*msb, the last bit in the stream now comes first*/
 int i;

 for(i=(ds->size - 1); i>=0; i--)
  printf("%d", get_bit(ds, i));

 printf("\n");
}

void print_arctic_data(struct data_stream *ds, int size, int arctics[])
{
 /*print each bit of stream, in binary*/
 /*removing the bypass bit for bypassed*/
 /*arctics and separating the arctics data*/
 int i, j;
 int place=0;

 for (j=0; j<NA; j++)
  if (arctics[j] == 1)
   {
      printf("Arctic %d ", j);
      for(i=0; i<size; i++)
       printf("%d", get_bit(ds, place++));
      printf("\n");
```

```
    }
   else place++;
}

void fprint_arctic_data(struct data_stream *ds, int size, int arctics[])
{
 /*print each bit of stream, in binary*/
 /*removing the bypass bit for bypassed*/
 /*arctics and separating the arctics data*/
 int i, j;
 int place=0;

 for (j=0; j<NA; j++)
  if (arctics[j] == 1)
    {
        fprintf(logf, "Arctic %d ", j);
        for(i=0; i<size; i++)
          fprintf(logf, "%d", get_bit(ds, place++));
        fprintf(logf, "\n");
    }
   else place++;
}

int add_bits(struct data_stream *ds, int num_bits, int start,
             int bits)
/*add num_bits number of bits to ds starting at start (bit number)
 from the number bits*/
/*for now this is cludged using set byte and set bit*/
{
 int i;
 int j;


 for(i = 0; i<num_bits/8; i++)
  {
  set_byte(ds, i*8,
           (((255<<(i*8)) & bits)>> (i*8)));
 }
 j = (num_bits- (num_bits%8));

 for(i=0; i<num_bits%8; i++)
  set_bit(ds, j+i, ((1<<(j+i)) & bits)>> j+i);
```

```
}
struct data_stream *parse_to_ds(int data1,
                                int size1,
                                int arctics[NA])
/*creates a data stream with data1 for each specified
 arctic and no data for each bypasses*/
{
 struct data_stream *ds;
 int i, j,k;
 int count =0;
 int place =0;

 for(i=0; i<NA; i++)
  {
   if (arctics[i] == 1)
       count++;
  }

 ds = create_data_stream(count*size1);

 for(i=0; i<NA; i++)
  {
   if (arctics[i] == 1)
       {
        for(j=0; j<size1; j++)
         {
         set_bit(ds, place++, ((data1 & (1<<j)) >> j ));
         }
       }
  }

 return(ds);

}
```

D.4 scripts.h

```
int do_start_log(char log_name[], char level[]);
int do_end_log();
int do_read_psc();
int do_write_psc();
int do_reset();
int do_reset_controller();
int do_port_stat();
int do_port_stats();
int do_all_stats();
int do_clear_stats();
int do_errors();
int do_bufs_free();
int do_cfg_read_port();
int do_cfg_write_port();
int do_cfg_write_all();
int do_change_mode();
int do_extest();
int do_arcintest();
int do_do_llm_scan();

int get_next_line(char data[6][60]);

struct data_stream *read_arctic_and_data_lines(int
                                arctics[], int num_bits);

int read_arctic_lines(int arctics[]);

char *upcase(char *str);
```

D.5 scripts.c

```
/* Elth Ogston
   11-16-95
   Sample Scripting Lanugage to Interface with ANCIS commands

   Chris Ward
   Major additions and revisions
   4-96
*/


#include "ancis.h"
#include "scripts.h"

/*at present this scripting lanuage does NOT
  provide for $? input commands
  deal with FILENAME data replacement files
  allow for data check files


  It DOES
  allow each ANCIS command to be entered
  read commands and ARCTIC/data inputs
  allow for comment lines starting with a #

*/

/*global variables for these functions:  the script file to
  read from, log file pointer, log level, and the current line
  number being read */

  int line_number = 0;
  FILE *script;
  FILE *logf;
  int logf_level = 0;

#define COMMENT_CHAR '#'     /* character to denote comment line */

main(int argc, char *argv[])
{
  char line_parses[6][60];
```

```
/* check for the correct number of arguments to the run_script
  command (i.e. one, the file name) */

if (argc != 2)
 {
  printf("\nUsage:  %s <script file> \n", argv[0]);
  return 1;
 }

/*open the script file for reading*/

if ((script = fopen(argv[1], "r")) == NULL)
 {
  printf("can't open file %s\n", argv[1]);
  return 1;
 }


/* the main loop of the program goes through the script file lines
  one at a time, does a big if-then lookup of the command names
  and calls a do_that_command function.  Each do_command function
  reads the data/ARCTIC lines after the command (until it reaches
  a new command) and then calls the function itself with the given
  data.

  lines are always read with the get_next_line command
    this parses out the data on that line, be it a command name,
       input data, or ARCTIC number.  It retuns (as arguments)
       the data as a set of strings, in order, of the strings that were
       seperated by white space in the line.

       If the next line is the end of the file, get_next_line returns
       EOF, otherwise it returns TRUE.

       the longest possible data field is 60 bits.  Since C ints(longs)
       are 32 bits any data longer then 32 bits will be divided into
       two strings.  Data is written in the script files as hex numbers.

       max string size will be from file names.  These will be limited
       to 60 character length strings.  Lines never have more then
       3 strings on them.
       */
```

```
while(get_next_line(line_parses) != EOF)
 {
  /* yep, it's the great big lookup table of commands*/

  if (strcmp(line_parses[0], "start_log") == 0)
      do_start_log(line_parses[1], line_parses[2]);

  else if (strcmp(line_parses[0], "end_log") == 0)
      do_end_log();

  else if (strcmp(line_parses[0], "read_psc") == 0)
      do_read_psc();

  else if (strcmp(line_parses[0], "write_psc") == 0)
      do_write_psc();

  else if (strcmp(line_parses[0], "reset") == 0)
      do_reset();

  else if (strcmp(line_parses[0], "reset_controller") == 0)
      do_reset_controller();

  else if (strcmp(line_parses[0], "port_stat") == 0)
      do_port_stat();

   else if (strcmp(line_parses[0], "port_stats") == 0)
      do_port_stats();

  else if (strcmp(line_parses[0], "all_stats") == 0)
      do_all_stats();

  else if (strcmp(line_parses[0], "clear_stats") == 0)
      do_clear_stats();

  else if (strcmp(line_parses[0], "errors") == 0)
      do_errors();

  else if (strcmp(line_parses[0], "bufs_free") == 0)
      do_bufs_free();

  else if (strcmp(line_parses[0], "cfg_read_port") == 0)
```

```
        do_cfg_read_port();

    else if (strcmp(line_parses[0], "cfg_write_port") == 0)
        do_cfg_write_port();

    else if (strcmp(line_parses[0], "cfg_write_all") == 0)
        do_cfg_write_all();

    else if (strcmp(line_parses[0], "change_mode") == 0)
        do_change_mode();

    else if (strcmp(line_parses[0], "extest") == 0)
        do_extest();

    else if (strcmp(line_parses[0], "arcintest") == 0)
        do_arcintest();

    else if (strcmp(line_parses[0], "do_llm_scan") == 0)
        do_do_llm_scan();

    else {
        printf("syntax error or unrecognized function line %d; %s \n",
            line_number, &line_parses[0]);
        if (logf_level>0)
          fprintf(logf, "syntax error or unrecognized function line %d; %s \n",
                line_number, &line_parses[0]);
    }
   }

 /*close script file*/
}


int do_start_log(char logf_name[], char level[])
{
 if (strlen(level)>0)
  logf_level = atoi(level);
 else
  logf_level = 1;

 if(TEST>0)
  printf("\nstart_log %s, level: %d\n", logf_name, logf_level);
```

```
 if ((logf = fopen(logf_name, "w")) == NULL)
   {
    printf("can't open file %s\n", logf_name);
    logf_level = 0;
    return 1;
   }
 printf("Log file %s sucessfully opened, level %d.\n", logf_name, logf_level);
 fprintf(logf, "Begin log file %s at level %d.\n\n", logf_name, logf_level);

 return 0;
}

int do_end_log()
{
 fclose(logf);
 logf_level = 0;

  if(TEST>0)
    printf("\nLog file closed.\n");

 return 0;
}

int do_read_psc()
{
 int arctics[NA];
 int count;
 struct data_stream *ds;

 if(TEST>0)
   printf("\nread_psc\n");
 if(logf_level>0)
   fprintf(logf, "\nread_psc\n");

 count = read_arctic_lines(arctics);

 ds = create_data_stream(count*18);
 /*psc is 18 bits long*/

 read_psc(ds, arctics);

 if(TEST>0)
   {
```

```
    printf("psc read = ");
    print_stream(ds);
   }
 if(logf_level>0)
   {
    fprintf(logf, "psc read = ");
    fprint_stream(ds);
   }

 delete_data_stream(ds);
}

int do_write_psc()
{
 struct data_stream *data;
 int arctics[NA];
 int i;

 data = read_arctic_and_data_lines(arctics, 18);

 if(TEST>0)
   {
   printf("\nwrite_psc \n");
   printf("data = \n");
   print_stream(data);
   printf("arctics = ");
   for(i=0; i<NA; i++)
     printf("%d", arctics[i]);
   printf("\n \n");
 }
 if(logf_level>0)
   {
   fprintf(logf, "\nwrite_psc \n");
   fprintf(logf, "data = \n");
   fprint_stream(data);
   fprintf(logf, "arctics = ");
   for(i=0; i<NA; i++)
     fprintf(logf, "%d", arctics[i]);
   fprintf(logf, "\n \n");
 }

 write_psc(data, arctics);
```

```
  delete_data_stream(data);
}

int do_reset()
{
  int arctics[NA];
  int i;

  read_arctic_lines(arctics);

  if(TEST>0)
    {
    printf("\nreset \n");
    printf("arctics = ");
    for(i=0; i<NA; i++)
      printf("%d", arctics[i]);
    printf("\n \n");
  }
  if(logf_level>0)
    {
    fprintf(logf, "\nreset \n");
    fprintf(logf, "arctics = ");
    for(i=0; i<NA; i++)
      fprintf(logf, "%d", arctics[i]);
    fprintf(logf, "\n \n");
  }

  reset(arctics);
}

int do_reset_controller()
{
  if(TEST>0)
    printf("\nreset_controller \n");
  if(logf_level>0)
    fprintf(logf, "\nreset_controller \n");

  board_reset();
}

int do_port_stat()
{
  char next_line[6][60];
```

```
int arctics[NA];
int i, count;
struct data_stream *ds;
int port, clear=0;
enum STAT stat;

get_next_line(next_line);

port = atoi(next_line[0]);

upcase(next_line[1]);
if (!strcmp(next_line[1], "PACKETS"))
  stat = STAT_PACKETS;
else if (!strcmp(next_line[1], "PRIORITY"))
  stat = STAT_PRIORITY;
else if (!strcmp(next_line[1], "UP"))
  stat = STAT_UP;
else if (!strcmp(next_line[1], "DOWN"))
  stat = STAT_DOWN;
else if (!strcmp(next_line[1], "IDLE"))
  stat = STAT_IDLE;
else if (!strcmp(next_line[1], "WAIT"))
  stat = STAT_WAIT;

if (!strcmp(upcase(next_line[2]), "CLEAR"))
  clear = 1;

count = read_arctic_lines(arctics);
ds = create_data_stream(count*38);
/*reg is 36 + 2 status bits long*/

if(TEST>0)
  {
    printf("\nport_stat \n");
    printf("port %d \n", port);
    printf("stat 0x%x \n", stat);
    printf("clear %d \n", clear);
    printf("arctics = ");
    for(i=0; i<NA; i++)
        printf("%d", arctics[i]);
    printf("\n \n");
  }
if(logf_level>0)
```

```
    {
     fprintf(logf, "\nport_stat \n");
     fprintf(logf, "port %d \n", port);
     fprintf(logf, "stat 0x%x \n", stat);
     fprintf(logf, "clear %d \n", clear);
     fprintf(logf, "arctics = ");
     for(i=0; i<NA; i++)
          fprintf(logf, "%d", arctics[i]);
     fprintf(logf, "\n \n");
    }

  port_stat(port, stat, ds, clear, arctics);

  if(TEST>0)
    {
     printf("port_stat = ");
     print_stream(ds);
    }
  if(logf_level>0)
    {
     fprintf(logf, "port_stat = ");
     fprint_stream(ds);
    }

  delete_data_stream(ds);

}

int do_port_stats()
{
  char next_line[6][60];
  int arctics[NA];
  int i, count;
  struct data_stream *ds[6];
  int port, clear=0;

  get_next_line(next_line);
  port = atoi(next_line[0]);
  if (!strcmp(upcase(next_line[1]), "CLEAR"))
    clear = 1;

  count = read_arctic_lines(arctics);
  for(i=0; i<6; i++)
```

```
  ds[i] = create_data_stream(count*38);
 /*reg is 36 + 2 status bits long*/

 if(TEST>0)
  {
   printf("\nport_stats \n");
   printf("port %d \n", port);
   printf("clear %d \n", clear);
   printf("arctics = ");
   for(i=0; i<NA; i++)
       printf("%d", arctics[i]);
   printf("\n\n");
  }
 if(logf_level>0)
  {
   fprintf(logf, "\nport_stats \n");
   fprintf(logf, "port %d \n", port);
   fprintf(logf, "clear %d \n", clear);
   fprintf(logf, "arctics = ");
   for(i=0; i<NA; i++)
       fprintf(logf, "%d", arctics[i]);
   fprintf(logf, "\n\n");
  }

 port_stats(port, ds, clear, arctics);

 if(logf_level>0)
  {
   fprintf(logf, "port_stats = \n");
   for (i=0; i<6; i++)
       {
        fprintf(logf, "\t");
        fprint_stream(ds[i]);
       }
  }
 printf("port_stats = \n");
 for (i=0; i<6; i++)
  {
   printf("\t");
   print_stream(ds[i]);
   delete_data_stream(ds[i]);
  }
}
```

```
int do_all_stats()
{
 char next_line[6][60];
 int arctics[NA];
 int i, j, count;
 struct data_stream *ds[4][6];
 int clear=0;

 get_next_line(next_line);
 if (!strcmp(upcase(next_line[0]), "CLEAR"))
  clear = 1;

 count = read_arctic_lines(arctics);
 for(i=0; i<6; i++)
  for(j=0; j<4; j++)
   ds[j][i] = create_data_stream(count*38);
 /*reg is 36 + 2 status bits long*/

 if(TEST>0)
  {
   printf("\nall_stats \n");
   printf("clear %d\n", clear);
   printf("arctics = ");
   for(i=0; i<NA; i++)
       printf("%d", arctics[i]);
   printf("\n \n");
  }

 all_stats(ds, clear, arctics);

 if(logf_level>0)
  {
   fprintf(logf, "\nall_stats \n");
   fprintf(logf, "clear %d\n", clear);
   fprintf(logf, "arctics = ");
   for(i=0; i<NA; i++)
       fprintf(logf, "%d", arctics[i]);
   fprintf(logf, "\n \n");
   fprintf(logf, "all_stats = \n");
   for(j=0; j<4; j++)
       {
        fprintf(logf, "  port %d\n", j);
```

```
            for (i=0; i<6; i++)
              {
               fprintf(logf, "\t");
               fprint_stream(ds[j][i]);
              }
            }
    }

  printf("all_stats = \n");
  for(j=0; j<4; j++)
    {
     printf("  port %d\n", j);
     for (i=0; i<6; i++)
          {
           printf("\t");
           print_stream(ds[j][i]);
           delete_data_stream(ds[j][i]);
          }
    }
}

int do_clear_stats()
{
  int arctics[NA];

  read_arctic_lines(arctics);

  if(TEST>0)
    printf("\nclear_stats \n");
  if(logf_level>0)
    fprintf(logf, "\nclear_stats \n");

  clear_stats(arctics);
}

int do_errors()
{
  char next_line[6][60];
  int arctics[NA];
  int i, count;
  struct data_stream *ds;
  int clear=0;
```

```
 get_next_line(next_line);
 if (!strcmp(upcase(next_line[0]), "CLEAR"))
  clear = 1;

 count = read_arctic_lines(arctics);
 ds = create_data_stream(count*60);
 /*one register, 60 bits long.  no status required*/

 if(TEST>0)
  {
   printf("\nerrors \n");
   printf("clear %d \n", clear);
   printf("arctics = ");
   for(i=0; i<NA; i++)
        printf("%d", arctics[i]);
   printf("\n \n");
  }
 if(logf_level>0)
  {
   fprintf(logf, "\nerrors \n");
   fprintf(logf, "clear %d \n", clear);
   fprintf(logf, "arctics = ");
   for(i=0; i<NA; i++)
        fprintf(logf, "%d", arctics[i]);
   fprintf(logf, "\n \n");
  }

 errors(ds, clear, arctics);

 if(TEST>0)
  {
   printf("errors = \n");
   print_stream(ds);
  }
 if(logf_level>0)
  {
   fprintf(logf, "errors = \n");
   fprint_stream(ds);
  }

 delete_data_stream(ds);
}
```

```
int do_bufs_free()
{
 int arctics[NA];
 int i, count;
 struct data_stream *ds;

 count = read_arctic_lines(arctics);
 ds = create_data_stream(count*20);
 /*one register, 20 bits long. no status*/

 if(TEST>0)
  {
   printf("\nbufs_free \n");
   printf("arctics = ");
   for(i=0; i<NA; i++)
       printf("%d", arctics[i]);
   printf("\n \n");
  }
 if(logf_level>0)
  {
   fprintf(logf, "\nbufs_free \n");
   fprintf(logf, "arctics = ");
   for(i=0; i<NA; i++)
       fprintf(logf, "%d", arctics[i]);
   fprintf(logf, "\n \n");
  }

 bufs_free(ds, arctics);

 if(TEST>0)
  {
   printf("bufs_free = ");
   print_stream(ds);
  }
 if(logf_level>0)
  {
   fprintf(logf, "bufs_free = ");
   fprint_stream(ds);
  }

 delete_data_stream(ds);
}
```

```
int do_cfg_read_port()
{
 char next_line[6][60];
 int arctics[NA];
 int i, count;
 struct data_stream *ds[5];
 int port;

 get_next_line(next_line);
 port = atoi(next_line[0]);

 count = read_arctic_lines(arctics);
 for(i=0; i<5; i++)
  ds[i] = create_data_stream(count*34);
 /*reg is 32 + 2 status bits long*/

 if(TEST>0)
  {
   printf("\ncfg_read_port \n");
   printf("port %d \n", port);
   printf("arctics = ");
   for(i=0; i<NA; i++)
        printf("%d", arctics[i]);
   printf("\n \n");
  }
 if(logf_level>0)
  {
   fprintf(logf, "\ncfg_read_port \n");
   fprintf(logf, "port %d \n", port);
   fprintf(logf, "arctics = ");
   for(i=0; i<NA; i++)
        fprintf(logf, "%d", arctics[i]);
   fprintf(logf, "\n \n");
  }

 cfg_read_port(port, ds, arctics);

 if(logf_level>0)
  {
   fprintf(logf, "cfg_read_port = \n");
   for (i=0; i<5; i++)
        {
         fprintf(logf, "\t");
```

```
            fprint_stream(ds[i]);
           }
    }

  printf("cfg_read_port = \n");
  for (i=0; i<5; i++)
    {
     printf("\t");
     print_stream(ds[i]);
     delete_data_stream(ds[i]);
    }
}

int do_cfg_write_port()
{
  struct data_stream *data[5];
  int arctics[NA];
  int i, port;
  char next_line[6][60];

  get_next_line(next_line);
  port = atoi(next_line[0]);

  for(i=0; i<5; i++)
    {
     get_next_line(next_line);
     data[i] = read_arctic_and_data_lines(arctics, 34);
    }

  if(TEST>0)
    {
    printf("\ncfg_write_port \n");
    printf("port %d \n", port);
    printf("data = \n");
    for(i=0; i<5; i++)
      print_stream(data[i]);
    printf("arctics = ");
    for(i=0; i<NA; i++)
      printf("%d", arctics[i]);
    printf("\n \n");
  }
  if(logf_level>0)
    {
```

```
   fprintf(logf, "\ncfg_write_port \n");
   fprintf(logf, "port %d \n", port);
   fprintf(logf, "data = \n");
   for(i=0; i<5; i++)
     fprint_stream(data[i]);
   fprintf(logf, "arctics = ");
   for(i=0; i<NA; i++)
     fprintf(logf, "%d", arctics[i]);
   fprintf(logf, "\n \n");
  }

  cfg_write_port(port, data, arctics);

  for(i=0; i<5; i++)
    delete_data_stream(data[i]);
}

int do_cfg_write_all()
{
  struct data_stream *data[5];
  int arctics[NA];
  int i;
  char next_line[6][60];

  for(i=0; i<5; i++)
    {
     get_next_line(next_line);
     data[i] = read_arctic_and_data_lines(arctics, 34);
  }

  if(TEST>0)
    {
   printf("\ncfg_write_all \n");
   printf("data = \n");
   for(i=0; i<5; i++)
     print_stream(data[i]);
   printf("arctics = ");
   for(i=0; i<NA; i++)
     printf("%d", arctics[i]);
   printf("\n \n");
  }
  if(logf_level>0)
    {
```

```
  fprintf(logf, "\ncfg_write_all \n");
  fprintf(logf, "data = \n");
  for(i=0; i<5; i++)
    fprint_stream(data[i]);
  fprintf(logf, "arctics = ");
  for(i=0; i<NA; i++)
    fprintf(logf, "%d", arctics[i]);
  fprintf(logf, "\n \n");
 }

 cfg_write_all(data, arctics);

 for(i=0; i<5; i++)
  delete_data_stream(data[i]);
}

int do_change_mode()
{
 int arctics[NA];
 char next_line[6][60];
 int i;

 get_next_line(next_line);
 read_arctic_lines(arctics);

 if(TEST>0)
  {
  printf("\nchange_mode \n");
  printf("%s \n", next_line[0]);
  printf("arctics = ");
  for(i=0; i<NA; i++)
    printf("%d", arctics[i]);
  printf("\n \n");
 }
 if(logf_level>0)
  {
  fprintf(logf, "\nchange_mode \n");
  fprintf(logf, "%s \n", next_line[0]);
  fprintf(logf, "arctics = ");
  for(i=0; i<NA; i++)
    fprintf(logf, "%d", arctics[i]);
  fprintf(logf, "\n \n");
 }
```

```
 if(strcmp(next_line[0], "normal") == 0)
  change_mode(MODE_normal, arctics);
 else if (strcmp(next_line[0], "config") == 0)
  change_mode(MODE_config, arctics);
 else if (strcmp(next_line[0], "low_level_test") == 0)
  change_mode(MODE_low_level_test, arctics);
 else
  {
   printf("unrecognized mode %s \n", next_line[0]);
   if(logf_level>0)
       fprintf(logf, "\nunrecognized mode %s \n", next_line[0]);
  }
}

int do_extest()
{
}

/*************************************************************/
int do_arcintest()
{
 int arctics[NA];
 char next_line[6][60];
 char temp_string[301];
 FILE *infile, *outfile;
 int i, n;

 get_next_line(next_line);
 read_arctic_lines(arctics);

 /* open the data files */
 if ((infile = fopen(next_line[0], "r")) == NULL)
  {
   printf("can't open file %s\n", next_line[0]);
   if(logf_level>0)
       fprintf(logf, "can't open file %s\n", next_line[0]);
   return 1;
  }
 if ((outfile = fopen(next_line[1], "w")) == NULL)
  {
   printf("can't open file %s\n", next_line[1]);
   if(logf_level>0)
```

```
        fprintf(logf, "can't open file %s\n", next_line[1]);
    return 1;
   }

 /* read n, the number of patterns, from the first line of the input file */
 if( fgets(temp_string, 290, infile) == NULL)
   {
    printf("empty input file %s\n", next_line[0]);
    if(logf_level>0)
        fprintf(logf, "empty input file %s\n", next_line[0]);
    return 1;
   }
 /* it will be in HEX */
 sscanf(temp_string, "%x", &n);

 if(TEST>0)
   {
    printf("arcintest \n");
    printf("input file: %s, output file: %s\n", next_line[0], next_line[1]);
    printf("%d patterns  (%x in hex)\n", n, n);
    printf("arctics = ");
    for(i=0; i<NA; i++)
        printf("%d", arctics[i]);
    printf("\n \n");
   }
  if(logf_level>0)
   {
    fprintf(logf, "arcintest \n");
    fprintf(logf, "input file: %s, output file: %s\n", next_line[0], next_line[1]);
    fprintf(logf, "%d patterns  (%x in hex)\n", n, n);
    fprintf(logf, "arctics = ");
    for(i=0; i<NA; i++)
        fprintf(logf, "%d", arctics[i]);
    fprintf(logf, "\n \n");
   }

 arcintest(infile, outfile, n, arctics);

 fclose(infile);
 fclose(outfile);

}
/*************************************************************/
```

```
int do_do_llm_scan()
{
}

/*
   lines are always read with the get_next_line command
     this parses out the data on that line, be it a command name,
        input data, or ARCTIC number. It retuns (as arguments)
        the data as a set of strings, in order, of the strings that were
        seperated by white space in the line.

        If the next line is the end of the file, get_next_line returns
        EOF, otherwise it returns TRUE.

        the longest possible data field is 60 bits. Since C ints(longs)
        are 32 bits any data longer then 32 bits will be divided into
        two strings. Data is written in the script files as hex numbers.

        max string size will be from file names. These will be limited
        to 60 character length strings. Lines never have more then
        3? strings on them.
*/
int get_next_line(char data[6][60])
{
 /*read in the next line*/
 /*parse out the stings within it, assumes there are only 3 data objects*/
 /*returns the number of data objects read, or EOF if the end of file
   was reached*/

 int i;
 char temp_string[301];

 /*initialize data*/
 for(i=0; i<6; i++)
   data[i][0] = '\0';

 i = 0;
 /*if the line was blank or a comment, ignore it*/
 while( (i <= 0) || (temp_string[0] == COMMENT_CHAR) )
   {
    /*get next line up to end line*/
```

```
    if( fgets(temp_string, 290, script) == NULL)
        return(EOF);

    /*parse it*/
    i = sscanf(temp_string, "%s %s %s %s %s %s", &data[0], &data[1],
                &data[2], &data[3], &data[4], &data[5]);

    /*update line count*/
    line_number++;
   }

 /*return number of data objects read*/

 return(i);
}

struct data_stream *read_arctic_and_data_lines(int
                                arctics[], int num_bits)
{
/*  read in ARCTICS lines and data to go with each
   each Arctic used gets one line, Arctics are listed
   in increasing order, data for each arctic follows
   on line immediately after the arctic.

   read_arctic_and_data_lines, returns number of arctics read,
   returns as args arctics list and pointer to data_stream

   prints an error if no Arctic lines are found
*/
 /*read arctic lines (up to NA of them) and
   one data line per arctic line*/

 struct data_stream *data;
 int temp_data[NA];
 char next_line[6][60];
 int i;
 int c;
 int count = 0 ;

 /*initialize arctics array*/
 for(i=0; i<NA; i++)
    arctics[i] = 0;
```

```
 /*read arctic and data lines*/
 for(i=0; i<NA; i++)
  {
   /*if next line isn't an arctics line, stop here*/
   /*an arctic line will always start with an A
       nothing else will (except data which would be
       proceeded by an arctic line)??*/

   if((c = getc(script)) != 'A')
       {
        ungetc(c, script);
        break;
       }
   ungetc(c, script);

   /*read Arctic line*/
   get_next_line(next_line);
   i = strtol(next_line[1], (char**)NULL, 16);
   if(i >= NA) {
        printf("error: arctic number larger then scan chain
          entered line number %d \n", line_number);
        if (logf_level>0)
         fprintf(logf, "error: arctic number larger then scan chain
               entered line number %d \n", line_number);
       }
   arctics[i] = 1;
   /*read data line*/
   get_next_line(next_line);
   temp_data[count] = strtol(next_line[0], (char**)NULL, 16);
   count++;
  }

 /*create data stream*/
 data = create_data_stream(count*num_bits);
 /*add data to data stream*/
 for(i=0; i<count; i++)
  add_bits(data, num_bits, i*num_bits, temp_data[i]);
 return(data);
}


int read_arctic_lines(int arctics[])
/*same as for read_arctic_and_data_lines, except for functions like
```

```
reads that don't take any data*/
{
/*  read in ARCTICS lines assuming no data follows
    each Arctic used gets one line, Arctics are listed
    in increasing order

    read_arctic_lines, returns number of arctics read,
    returns as args arctics list

    prints an error if no Arctic lines are found
*/
  /*read arctic lines (up to NA of them) and
    one data line per arctic line*/

  char next_line[6][60];
  int i;
  int c;
  int count = 0;

  /*initialize arctics array*/
  for(i=0; i<NA; i++)
     arctics[i] = 0;

  /*read arctic*/
  for(i=0; i<NA; i++)
    {
     /*if next line isn't an arctics line, stop here*/
     /*an arctic line will always start with an A
          nothing else will (except data which would be
          proceeded by an arctic line)??*/

     if((c = getc(script)) != 'A')
          {
           ungetc(c, script);
           break;
          }
     ungetc(c, script);

     /*read Arctic line*/
     get_next_line(next_line);
     i = strtol(next_line[1], (char**)NULL, 16);
     if(i >= NA) {
          printf("error: arctic number larger then scan chain
```

```
            entered line number %d \n", line_number);
          if (logf_level>0)
            fprintf(logf, "error: arctic number larger then scan chain
                  entered line number %d \n", line_number);
        }
    else {
        arctics[i] = 1;
        count++;
    }
   }

 return(count);
}


/* utility function */
/* changes each lower case letter in the input string to upper case */
/* modfies the input string. also returns a pointer to it */
char *upcase (char *str)
{
 char *temp = str;

 for(; *str!='\0'; str++)
  if (islower(*str))
    *str -= 32;
 return(temp);
}
```

D.6 upper_level.h

```
#define BOUNDARY_SIZE 323  /* number of cells in the boundary scan ring */

enum STAT {
  PACKETS = STAT_PACKETS,
  PRIORITY = STAT_PRIORITY,
  UP = STAT_UP,
  DOWN = STAT_DOWN,
  IDLE = STAT_IDLE,
  WAIT= STAT_WAIT};


void read_psc(struct data_stream *data, int arctics[]);

void write_psc(struct data_stream *data, int arctics[]);

void reset(int arctics[]);

void reset_controller();

void port_stat(int port, enum STAT stat, struct data_stream *data,
               int clear, int arctics[]);

void port_stats(int port, struct data_stream *data[6], int clear, int
                  arctics[]);

void all_stats(struct data_stream *data[4][6], int clear, int arctics[]);

void clear_stats(int arctics[]);

void errors(struct data_stream *data, int clear, int arctics[]);

void bufs_free(struct data_stream *data, int arctics[]);

void cfg_read_port(int port, struct data_stream *data[5],
                     int arctics[]);

void cfg_write_port(int port, struct data_stream *data[5],
                     int arctics[]);

void cfg_write_all(struct data_stream *data[5], int arctics[]);
```

```
void change_mode(enum MODE mode, int arctics[]);

void extest(FILE *data_in, FILE *data_out, int size, int arctics[]);

void arcintest(FILE *data_in, FILE *data_out, int size, int arctics[]);

void do_llm_scan(FILE *data_in, FILE *data_out, int arctics[]);
```

D.7 upper_level.c

```
#include "ancis.h"

void old_change_mode(enum MODE mode, int arctics[]);
int Global_Mode[NA];

extern  FILE *logf;             /* for log files */
extern  int logf_level;

void read_psc(struct data_stream *data, int arctics[])
{
 int i , len=0;

 if(TEST>1)
  {
   printf("read psc \n");
  }
 if(logf_level>1)
  {
   fprintf(logf, "read psc \n");
  }

 /*check for correct length data stream*/
 for(i=0; i<NA; i++)
  if (arctics[i] == 1)
   len++;

 if (get_ds_length(data) != ((NA-len) + len*18))
   {
       printf("wrong data stream length given to read_psc\n");
       if (logf_level>0)
         fprintf(logf, "wrong data stream length given to read_psc\n");
       return;
   }

 read_reg(CONTROL_PORT_STATS, arctics);
 get_status(data, 18, arctics, 0);
}

void write_psc(struct data_stream *data, int arctics[])
{
```

```
 if(TEST>1)
  {
   printf("write psc data:");
   print_stream(data);
  }
 if(logf_level>1)
  {
   fprintf(logf, "write psc data:");
   fprint_stream(data);
  }

 write_reg(CONTROL_PORT_STATS, data, 18, arctics);
 /*no status required*/
}


void reset(int arctics[])
{
 /*reset is done by writting the one bit long reset register
  0x341*/
 int i;
 struct data_stream *ds;

 /*create a junk data_stream to write*/
 ds = create_data_stream(NA);

 write_reg(SYSTEM_RESET, ds, 1, arctics);
 /*no status required*/

 /*deallocate data stream memory*/
 delete_data_stream(ds);

 /* wait 12 tck cycles */
 delay(12);

 /*set Global_Mode to configuration*/
 for(i=0; i<NA; i++)
  Global_Mode[i] = MODE_config;
}

void reset_controller()
{
```

```
 if(TEST>1)
  printf("reset_controller\n");
 if(logf_level>1)
  fprintf(logf, "reset_controller\n");

 /*reset the controller and cycle all the TAP controllers
  back to RTI
  uses board level functions*/

 board_reset;
}

void port_stat(int port, enum STAT stat, struct data_stream *data,
               int clear, int arctics[])
{

/*stats start at 280 +1 for each one, +6 for each port*/
/* +0x40 to clear */
/*stats do need status*/
/*register size is 36 + two status bits*/

 read_reg(stat+(6*port) + (0x40*clear), arctics);
 get_status(data, 38, arctics, 1);
}

void port_stats(int port, struct data_stream *data[6], int clear,
                  int arctics[])
{
 int i;

 for(i=0; i<6; i++)
  {
   read_reg(STAT_PACKETS+i+(6*port)+ (0x40*clear), arctics);
   get_status(data[i], 38, arctics, 1);
  }
}

void all_stats(struct data_stream *data[4][6], int clear, int arctics[])
{
 int i, j;

 for(j=0; j<4; j++)
  for(i=0; i<6; i++)
```

```
    {
        read_reg(STAT_PACKETS+i+(6*j)+(0x40*clear), arctics);
        get_status(data[j][i], 38, arctics, 1);
    }
}

void clear_stats(int arctics[])
{
 /*stats are cleared by reading the clear stats register
   size 36, + 2 status, status is needed*/

 struct data_stream *temp_ds;
 int len = 0;
 int i;

 if(TEST>1)
   printf("clear_stats \n");
 if(logf_level>1)
   fprintf(logf, "clear_stats \n");

 for(i=0; i<NA; i++)
   if(arctics[i] == 1)
     len++;

 len = (len*38)+ NA-len;
 temp_ds = create_data_stream(len);

 read_reg(STAT_CLEAR_ALL, arctics);
 get_status(temp_ds, 38, arctics, 1);

 delete_data_stream(temp_ds);
}

void errors(struct data_stream *data, int clear, int arctics[])
{
 /*one register, 60 bits long.  no status required*/
 /*to clear read ERROR+1*/


   read_reg(ERROR + clear, arctics);
   get_status(data, 60, arctics, 0);
}
```

```
void bufs_free(struct data_stream *data, int arctics[])
{
 /*one register, 20 bits long. no status*/
 read_reg(BF_COUNTERS, arctics);
 get_status(data, 20, arctics, 0);
}

void cfg_read_port(int port, struct data_stream *data[5],
                   int arctics[])
{
 /*each cfg register is 32 bits, status is needed*/
 /*registers start at 0, + one for next in port,
  + 40(hex) for each port*/

 /*you must be in configuration mode before running
  any of the configuration commands*/

 int i;

 /*check to make sure the mode is ok*/

 for(i=0; i<NA; i++)
  if((arctics[i] == 1) &
    (Global_Mode[i] != MODE_config))
    {
       printf("ERROR, attempt to access configuration registers when not in config
mode\n");
       if (logf_level>0)
         fprintf(logf, "ERROR, attempt to access configuration registers when not in
config mode\n");
       return;
    }

 for(i=0; i<5; i++)
  {
   read_reg(CONFIG_PORT + (0x40*port) + i, arctics);
   get_status(data[i], 34, arctics, 1);
  }
}

void cfg_write_port(int port, struct data_stream *data[5],
                   int arctics[])
{
```

```
 int i;
 struct data_stream *temp_ds;
 int len =0;

 /*check to make sure the mode is ok*/

 for(i=0; i<NA; i++)
  if((arctics[i] == 1) &
    (Global_Mode[i] != MODE_config))
    {
        printf("ERROR, attempt to access configuration registers when not in config
mode\n");
        if (logf_level>0)
          fprintf(logf, "ERROR, attempt to access configuration registers when not in
config mode\n");
        return;
    }

 for(i=0; i<NA; i++)
  if(arctics[i] == 1)
   len++;

 len = (len*34)+ NA-len;
 temp_ds = create_data_stream(len);

 for(i=0; i<5; i++)
  {
   write_reg(CONFIG_PORT + (0x40*port) + i, data[i], 34, arctics);
   get_status(temp_ds, 34, arctics, 1);
  }
 delete_data_stream(temp_ds);
}

void cfg_write_all(struct data_stream *data[5], int arctics[])
{
 /*write to the CONFIG_BROADCAST reg*/
 int i;
 struct data_stream *temp_ds;
 int len =0;

 /*check to make sure the mode is ok*/

 for(i=0; i<NA; i++)
```

```
  if((arctics[i] == 1) &
    (Global_Mode[i] != MODE_config))
    {
        printf("ERROR, attempt to access configuration registers when not in config
mode\n");
        if (logf_level>0)
          fprintf(logf, "ERROR, attempt to access configuration registers when not in
config mode\n");
        return;
    }

 for(i=0; i<NA; i++)
  if(arctics[i] == 1)
   len++;

 len = (len*34)+ NA-len;
 temp_ds = create_data_stream(len);

 for(i=0; i<5; i++)
  {
   write_reg(CONFIG_BROADCAST + i, data[i], 34, arctics);
   get_status(temp_ds, 34, arctics, 1);
  }
 delete_data_stream(temp_ds);
}

void change_mode(enum MODE mode, int arctics[])
{
 /*must disable all inputs and outputs using psc
  before entering configuration mode*/
 /*mode changed by writting the M_reg, 2 bits,
  it's an access macro, not an arctic access op.*/
 /*Global_Mode keeps track of what mode each arctic
  is in*/

/* FOR NOW CHANGE MODE DOES NOT DEAL WITH ENABLING
 OR DISABLING INPUT PORTS, IT IS UP TO THE USER TO
 DEAL WITH THESE ARCTIC RULES */

 int i;

 if (TEST>1)
  printf("CHANGE MODE %d \n", mode);
```

```
  if (logf_level>1)
    fprintf(logf, "CHANGE MODE %d \n", mode);

  write_Mreg(mode, arctics);

  /*correct the Global_Mode*/

  for(i=0; i<NA; i++)
    if(arctics[i] == 1)
      Global_Mode[i] = mode;
}

void old_change_mode(enum MODE mode, int arctics[])
{
    /*must disable all inputs and outputs using psc
    before entering configuration mode*/
  /*mode changed by writting the M_reg, 2 bits,
    it's an access macro, not an arctic access op.*/
  /*Global_Mode keeps track of what mode each arctic
    is in*/

/*VERSION OF CHANGE MODE THAT ATTEMPTS TO DEAL WITH
INPUT PORTS, NOT DEBUGGED YET*/
  int i,j;
  struct data_stream *temp_ds;
  int len =0;
  int place =0;

  for(i=0; i<NA; i++)
    if(arctics[i] == 1)
      len++;

printf("CHANGE MODE %d \n", mode);

  /*psc is 18 ibts*/

  len = (len*18)+ NA-len;
  temp_ds = create_data_stream(len);

  if (mode == MODE_config)
    {
      for(i=0; i<NA; i++)
          if(arctics[i] == 1)
```

```
        {
         /*disable inputs outputs*/
         /*read current psc*/
         read_psc(temp_ds, arctics);
         /*set disable bits*/

         for(j=0; j<NA; j++)
          {
              if(arctics[j] == 0)
               place++;
              else
               {
                set_bit(temp_ds, place, ONE);
                set_bit(temp_ds, place+1, ONE);
                set_bit(temp_ds, place+4, ONE);
                set_bit(temp_ds, place+5, ONE);
                set_bit(temp_ds, place+8, ONE);
                set_bit(temp_ds, place+9, ONE);
                set_bit(temp_ds, place+13, ONE);
                set_bit(temp_ds, place+14, ONE);
                place = place + 18;
               }
          }

         /*write new psc*/
         write_psc(temp_ds, arctics);
        }
 }

write_Mreg(mode, arctics);

/*if modes was config, and changed to normal or llm_test reenable
  inputs and outputs.*/

if(mode != MODE_config)
 {
  for(i=0; i<NA; i++)
      if ((Global_Mode[i] == MODE_config) &
         (arctics[i] == 1))
        {
         /*enable inputs outputs*/
         /*read current psc*/
         read_psc(temp_ds, arctics);
```

```
            /*set enable bits*/

            for(j=0; j<NA; j++)
             {
                 if(arctics[j] == 0)
                  place++;
                 else
                  {
                    set_bit(temp_ds, place, ZERO);
                    set_bit(temp_ds, place+1, ZERO);
                    set_bit(temp_ds, place+4, ZERO);
                    set_bit(temp_ds, place+5, ZERO);
                    set_bit(temp_ds, place+8, ZERO);
                    set_bit(temp_ds, place+9, ZERO);
                    set_bit(temp_ds, place+13, ZERO);
                    set_bit(temp_ds, place+14, ZERO);
                    place = place + 18;
                  }
             }

            /*write new psc*/
            write_psc(temp_ds, arctics);
           }
    }

  /*correct the Global_Mode*/

  for(i=0; i<NA; i++)
   if(arctics[i] == 1)
     Global_Mode[i] = mode;

  delete_data_stream(temp_ds);
}

void extest(FILE *data_in, FILE *data_out, int size, int arctics[])
{

}

/*************************************************************************/
/***** int size --> the number of patterns  (should not be zero)   *****/
/***** This function is based on boundary scan ring information in  *****/
/*****  /home/prj4/arcticdf/arctic22df/control_test/mrings/bsc.def  *****/
```

```
/* NOTE: the file formats were originally designed with the opposite
  bit order in mind. The quick fix found here consists of reversing
  the direction of the data streams (reversing the bit order) at
  appropriate times.
  Also, because of the bit ordering in the data steam abstraction,
  bytes are reversed between the files and the data stream
*/

void arcintest(FILE *data_in, FILE *data_out, int size, int arctics[])
{
 char temp_string[301];
 int i, j, k, x;
 long int bit;
 struct data_stream *ds, *original_ds;
 int data[8];
 short sdata;

 load_JTAGIR(JTAG_arcintest, JTAG_bypass, arctics);

 /* create the data stream */
 ds = create_data_stream(BOUNDARY_SIZE);
 original_ds = create_data_stream(BOUNDARY_SIZE);

 /* read, scan, and write for each pattern */
 for(i=0; i <= size; i++)
  {
   /*** read data from the input file into the data stream*/
    /* last time through is just to get output from prev pattern */
   if (i == size)
       /* last time through, replace the original data */
       for(j=0; j<BOUNDARY_SIZE; j++)
         set_bit(ds, j, get_bit(original_ds, j));
   else
       /* if it's not the last time through, read from the infile */
       for(bit=0, k=0; k<5; k++)    /*for each line(max 8 bytes)*/
         {
          /*get next line up to end line*/
          if( fgets(temp_string, 290, data_in) == NULL)
           {
               printf("\n** Error: unexpected EOF in arcintest\n\n");
               if (logf_level>0)
                 fprintf(logf, "\n** Error: unexpected EOF in arcintest\n\n");
```

```
            return;
        }

    /*parse it*/
    sscanf(temp_string, "%x %x %x %x %x %x %x %x",
            &data[0], &data[1], &data[2], &data[3],
            &data[4], &data[5], &data[6], &data[7]);

    /*add the data to the data stream*/
    if (k==0)
        {
            /* first time is the psc */
            /* each byte has to be flipped, so MSB begins stream */
            set_byte(ds, bit, flip_byte((short)data[0]<<2));/*psc 6 bits*/
            bit+=6;
            set_byte(ds, bit, flip_byte((short)data[1]));  /*psc 8 bits*/
            bit+=8;
            set_byte(ds, bit, flip_byte((short)data[2]<<4));/*psc 4 bits*/
            bit+=4;
            bit++;  /*skip next bit, error_q, (leave it 0 -- don't care)*/
        }
    else
        {
            /* otherwise input port (frame_ipx, bf2, data_ipx) */
            bit+=3;   /*skip next 3 bits (leave them 0 -- don't care)*/
            set_byte(ds, bit++, ((short)data[0]));    /*frame_ipx 1 bit*/
            bit++;      /*skip next bit (leave it 0 -- don't care)*/
            set_byte(ds, bit++, ((short)data[1]));    /*bf2 1 bit*/
            set_byte(ds, bit, flip_byte((short)data[2])); /*data byte*/
            bit+=8;
            set_byte(ds, bit, flip_byte((short)data[3])); /*data byte*/
            bit+=8;
            set_byte(ds, bit, flip_byte((short)data[4])); /*data byte*/
            bit+=8;
            set_byte(ds, bit, flip_byte((short)data[5])); /*data byte*/
            bit+=8;
        }
    }

flip_stream(ds);
/*** shift the data into the boundary scan cells */
if (i == 0)
    load_JTAGDR2_beg(ds, arctics, BOUNDARY_SIZE);
```

```
else
    load_JTAGDR2(ds, arctics, BOUNDARY_SIZE);  /* last pattern */

flip_stream(ds);

/*** write return data from the data stream to the ouput file */
 /* don't write output for the first pattern */
if (i == 0)
    { /* first time through save the data, but don't output */
     for(j=0; j<BOUNDARY_SIZE; j++)
       set_bit(original_ds, j, get_bit(ds, j));
    }
else
    {
     bit = 0;
     get_byte(ds, bit, &sdata);     /*psc 6 bits*/
     bit+=6;
     fprintf(data_out, "%2x ", (flip_byte(sdata)>>2));
     get_byte(ds, bit, &sdata);     /*psc 8 bits*/
     bit+=8;
     fprintf(data_out, "%2x ", flip_byte(sdata));
     get_byte(ds, bit, &sdata);     /*psc 4 bits*/
     bit+=4;
     fprintf(data_out, "%x ", (flip_byte(sdata)>>4));
     get_byte(ds, bit++, &sdata);     /*error_q 1 bit*/
     fprintf(data_out, "%x\n", (sdata & 1));

     /* lines 2-5, the output ports */
     bit+=(4*38);     /*skip the next (4 ports x 38 bits each)  bits*/
     for (k=0; k<4; k++)
      {
       get_byte(ds, bit, &sdata);     /*data_shad 8 bits*/
       bit+=8;
       fprintf(data_out, "%2x ", flip_byte(sdata));
       get_byte(ds, bit, &sdata);     /*data_shad 8 bits*/
       bit+=8;
       fprintf(data_out, "%2x ", flip_byte(sdata));
       get_byte(ds, bit, &sdata);     /*data_shad 8 bits*/
       bit+=8;
       fprintf(data_out, "%2x ", flip_byte(sdata));
       get_byte(ds, bit, &sdata);     /*data_shad 8 bits*/
       bit+=8;
       fprintf(data_out, "%2x ", flip_byte(sdata));
```

```
            get_byte(ds, bit, &sdata);      /*next 4 bits*/
            bit+=4;
            fprintf(data_out, "%x %x %x %x\n", (sdata&1), ((sdata>>1)&1),
                                    ((sdata>>2)&1), ((sdata>>3)&1));
            bit+=2;     /*skip the next 2 bits*/
           }
        fprintf(data_out, "end pattern %d\n\n", i);
    }
   }

 /*** End the arcintest procedure */
 load_JTAGDR2_end();
 load_JTAGIR(JTAG_bypass, JTAG_bypass, arctics);

 /* delete the data stream */
 delete_data_stream(ds);
 delete_data_stream(original_ds);
 return;
}
/*************************************************************/

void do_llm_scan(FILE *data_in, FILE *data_out, int arctics[])
{
 /*** Not currently implemented ***/

 int i;
 struct data_stream *temp_ds;
 int len =0;
 int place;

 /*create a data stream to send things in on*/
 for(i=0; i<NA; i++)
  if(arctics[i] == 1)
    len++;

 /*llm scan reg is 6 bits*/
 len = (len*6)+ NA-len;
 temp_ds = create_data_stream(len);

 /*read in next stream*/

 for(i=0; i<len; i++)
  {
```

```
/*      set_bit(temp_ds, place, atoi(getc(data_in)));*/
    }

  /*send to arctics*/
  /*write out to file*/

  /*change mode*/
  change_mode(MODE_low_level_test, arctics);


  /*return mode to normal*/
  change_mode(MODE_normal, arctics);
}
```

D.8 arctic_access_ops.h

```
void read_reg (enum REGISTERS address, int arctics[]);

void write_reg (enum REGISTERS address, struct data_stream *ds,
                int data_len, int arctics[]);

int get_status (struct data_stream *ds, int data_len, int arctics[],
                int status);
```

D.9 arctic_access_ops.c

```
#include "ancis.h"

extern  FILE *logf;
extern  int logf_level;

void read_reg (enum REGISTERS address, int arctics[])
{
 if(TEST>2)
   printf("   read reg: %x\n", address);
 if(logf_level>2)
   fprintf(logf, "   read reg: %x\n", address);

 write_AOAreg(address, arctics);
 write_AOIreg(AOI_read_reg, arctics);
}

void write_reg (enum REGISTERS address, struct data_stream *ds,
                int data_len, int arctics[])
{
 if(TEST>2)
   {
    printf("   write reg addr: %x  data:", address);
    print_stream(ds);
   }
 if(logf_level>2)
   {
    fprintf(logf, "   write reg addr: %x  data:", address);
    fprint_stream(ds);
   }

 write_AOAreg(address, arctics);
 write_AOSandDreg(ds, arctics, data_len);
 write_AOIreg(AOI_write_reg, arctics);
}

int get_status (struct data_stream *ds, int data_len, int arctics[],
                int status)
{
 int place = 0;
 enum AOS_values current_status;
```

```
enum AOS_values total_status = IN_PROGRESS;
int i, j=0;

if(TEST>2)
 printf("  get status\n");
if(logf_level>2)
 fprintf(logf, "  get status\n");

read_AOSandDreg(ds, arctics, data_len);

if(status >= 1)
 /*status is needed, check it and continue to read 'till
  completed*/
 {

  while(total_status == IN_PROGRESS)
      {
       total_status = COMPLETED;
       place = 0;
       for(i=0; i<NA; i++)
        {
         if(arctics[i] == 1)
             {
              /*look at first two bits, well last
                two since it comes out backwards*/
              place = place + data_len -2;
              current_status =
                get_bit(ds, place++);
              current_status = current_status +
                get_bit(ds, place++)*2;

              if(current_status == IN_PROGRESS)
                total_status = IN_PROGRESS;
              else if ((current_status == MODE_ERROR)
                        || (current_status == BROADCAST_READ_ERROR))
                {
                 fprintf(logf, "\nERROR in get_status: %d\n\n",
                          current_status);
                 return(current_status);
                }
             }
        }
       if(total_status == IN_PROGRESS)
```

```
            read_AOSandDreg(ds, arctics, data_len);
        }
    }
 return(COMPLETED);
}
```

D.10 access_macs.h

```
enum AOS_values {
  COMPLETED = 0,
  IN_PROGRESS =1,
  MODE_ERROR =2,
  BROADCAST_READ_ERROR = 3};

enum REGISTERS{
  CONTROL_PORT_STATS = 0x200,
  SYSTEM_RESET = 0x341,
  STAT_PACKETS = 0x280,
  STAT_PRIORITY = 0x281,
  STAT_UP = 0x282,
  STAT_DOWN = 0x283,
  STAT_IDLE = 0x284,
  STAT_WAIT = 0x285,
  STAT_CLEAR_ALL = 0x2FF,
  ERROR = 0x300,
  BF_COUNTERS = 0x240,
  CONFIG_PORT = 0x000,
  CONFIG_BROADCAST = 0x1E0,
  MANUF_TEST_RING = 0x380};

enum INSTRS{
AOI_write_reg = 0,
AOI_read_reg = 1};


enum MODE {
MODE_config = 0,
MODE_normal = 1,
MODE_low_level_test = 2};


 void write_AOAreg(enum REGISTERS address,
                   int arctics[NA]);

 void write_AOSandDreg(struct data_stream *ds,
                       int arctics[NA], int data_len);

 void write_AOIreg(enum INSTRS instruction,
```

```
                int arctics[NA]);

void read_AOSandDreg(struct data_stream *ds,
                int arctics[NA], int data_len);

void write_Mreg(short mode, int arctics[NA]);

void set_up_llm_scan(struct data_stream *ds, int arctics[]);

void llm_scan(struct data_stream *ds,int arctics[]);

void end_llm_scan();
```

D.11 access_macs.c

```
#include "ancis.h"

extern  FILE *logf;
extern  int logf_level;

void write_AOAreg(enum REGISTERS address,
                  int arctics[NA])
{
 struct data_stream *ds;

 if(TEST>3)
  printf("      write AOAreg: %x \n", address);
 if(logf_level>3)
  fprintf(logf, "      write AOAreg: %x \n", address);

 ds = parse_to_ds(address, 10, arctics);

 load_JTAGIR(JTAG_sel_AOAreg, JTAG_bypass, arctics);
 load_JTAGDR(ds, arctics, 10);

 delete_data_stream(ds);
}

void write_AOIreg(enum INSTRS instruction,
                  int arctics[NA])
{
 struct data_stream *ds;

 if(TEST>3)
  printf("      write AOIreg instr: %x \n", instruction);
 if(logf_level>3)
  fprintf(logf, "      write AOIreg instr: %x \n", instruction);

 ds = parse_to_ds(instruction, 2, arctics);

 load_JTAGIR(JTAG_sel_AOIreg, JTAG_bypass, arctics);
 load_JTAGDR(ds, arctics, 2);

 delete_data_stream(ds);
}
```

```
void write_AOSandDreg(struct data_stream *ds, int arctics[NA],
                      int data_len)
{
 if(TEST>3)
   {
    printf("      write AOSandDreg data:");
    print_stream(ds);
   }
 if(logf_level>3)
   {
    fprintf(logf, "      write AOSandDreg data:");
    fprint_stream(ds);
   }

 load_JTAGIR(JTAG_sel_AOSandDreg, JTAG_bypass, arctics);
 load_JTAGDR(ds, arctics, data_len);
}

void write_Mreg(short mode, int arctics[NA])
{
 struct data_stream *ds;

 if(TEST>3)
   printf("      write Mreg \n");
 if(logf_level>3)
   fprintf(logf, "      write Mreg \n");

 ds = parse_to_ds(mode, 2, arctics);

 load_JTAGIR(JTAG_sel_Mreg, JTAG_bypass, arctics);
 load_JTAGDR(ds, arctics, 2);

 delete_data_stream(ds);
}

void read_AOSandDreg(struct data_stream *ds,
                     int arctics[NA], int data_len)
{
 if(TEST>3)
   printf("      read AOSandDreg \n");
 if(logf_level>3)
   fprintf(logf, "      read AOSandDreg \n");
```

```
 load_JTAGIR(JTAG_sel_AOSandDreg, JTAG_bypass, arctics);
 load_JTAGDR(ds, arctics, data_len);
}


void set_up_llm_scan(struct data_stream *ds, int arctics[])
{
 struct data_stream *temp_ds;

 temp_ds = parse_to_ds(MANUF_TEST_RING, 10, arctics);

 load_JTAGIR(JTAG_sel_AOAreg, JTAG_bypass, arctics);
 load_JTAGDR(temp_ds, arctics, 10);
 load_JTAGIR(JTAG_sel_AOSandDreg, JTAG_bypass, arctics);

 delete_data_stream(temp_ds);

 load_JTAGDR2_beg(ds, arctics, 6);
}

void llm_scan(struct data_stream *ds,int arctics[])
{
 load_JTAGDR2(ds, arctics, 6);
}

void end_llm_scan()
{
 load_JTAGDR2_end;
}
```

D.12 jtag_trans.h

```
enum JTAG_INST{
  JTAG_bypass = 0xF,
  JTAG_extest = 0x0,
  JTAG_preload = 0x1,
  JTAG_arcintest = 0x2,
  JTAG_sel_Mreg = 0x4,
  JTAG_sel_AOAreg = 0x5,
  JTAG_sel_AOIreg = 0x6,
  JTAG_sel_AOSandDreg = 0x7,
  JTAG_clamp = 0x8};

void load_JTAGIR(enum JTAG_INST instr, enum JTAG_INST alternate,
                 int arctics[NA]);

void load_JTAGDR(struct data_stream *ds, int arctics[NA],
                 int data_len);

void load_JTAGDR2_beg(struct data_stream *ds, int arctics[],
                 int data_length);

void load_JTAGDR2(struct data_stream *ds, int arctics[],
                 int data_length);

void load_JTAGDR2_end();
```

D.13 jtag_trans1.c

```
/* This is the jtag_trans.c file for testing with the Verilog model */

#include "ancis.h"

extern  FILE *logf;
extern  int logf_level;

void load_JTAGIR(enum JTAG_INST instr, enum JTAG_INST alternate,
                 int arctics[NA])
{
 struct data_stream *tms_ds, *tdi_ds;
 int place = 0 ;
 int i;

 if(TEST>4)
  printf("      load JTAGIR instr: %x \n", instr);
 if(logf_level>4)
  fprintf(logf, "      load JTAGIR instr: %x \n", instr);

 /* length of tdi and tms data streams: all jtag instructions
   are 4 bits long so length = NA*4 */

 tms_ds = create_data_stream((NA*4)+8);
 tdi_ds = create_data_stream((NA*4)+8);

 /*get to shift IR state from RTI*/
 set_bit(tms_ds, place++, ONE);
 set_bit(tms_ds, place++, ONE);
 set_bit(tms_ds, place++, ZERO);
 set_bit(tms_ds, place++, ZERO);


 for(i=0; i<NA; i++)
  {
   if(arctics[i] == 0)

       /*put in alternate*/
       {
        set_bit(tdi_ds, place++, (alternate & 1));
        set_bit(tdi_ds, place++, ((alternate>>1) & 1));
```

```
          set_bit(tdi_ds, place++, ((alternate>>2) & 1));
          set_bit(tdi_ds, place++, ((alternate>>3) & 1));
        }
    else
        /*put in instr*/
        {
          set_bit(tdi_ds, place++, (instr & 1));
          set_bit(tdi_ds, place++, ((instr>>1) & 1));
          set_bit(tdi_ds, place++, ((instr>>2) & 1));
          set_bit(tdi_ds, place++, ((instr>>3) & 1));
        }
  }

 place--;

 /*go to RTI*/
 set_bit(tms_ds, place++, ONE);
 set_bit(tms_ds, place++, ONE);
 set_bit(tms_ds, place++, ZERO);

 board_send_bits(tdi_ds, tms_ds);

 delete_data_stream(tms_ds);
 delete_data_stream(tdi_ds);

 /*boad interface needs to deal with sticky and lost
   bits at the end*/
}

void load_JTAGDR(struct data_stream *ds, int arctics[NA],
                 int data_len)
{
 struct data_stream *tms_ds, *tdi_ds;
 int place = 0;
 int ds_place = 0;
 int i, j;
 int len = 0;

 if(TEST>4)
   {
    printf("       load JTAGDR data:");
    print_stream(ds);
   }
```

```
if(logf_level>4)
  {
   fprintf(logf, "        load JTAGDR data:");
   fprint_stream(ds);
  }

/*figure out length of streams*/
for(i=0; i<NA; i++)
 if(arctics[i] == 1)
   len++;

len = len*data_len + (NA-len) + 6;

/*create streams*/
tms_ds = create_data_stream(len);
tdi_ds = create_data_stream(len);


/*get to shift DR state from RTI*/
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ZERO);
set_bit(tms_ds, place++, ZERO);

/*shift DR*/
for(i=0; i<NA; i++)
 if(arctics[i] == 0)
   /*bypass*/
   place++;
 else
   {
       /*copy in data*/
       for(j=0; j<data_len; j++)
         set_bit(tdi_ds, place++, get_bit(ds, ds_place++));
   }

place--;

/*go to RTI*/
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ZERO);

board_send_bits(tdi_ds, tms_ds);
```

```
 /*bits returnd in tdi_ds, remove extra bits, and
   send back in ds,
   3 extra
   1 for each bypass
   data_len for each actual
   3 more extra*/

 place = 3;
 ds_place = 0;
 for(i=0; i<NA; i++)
   if(arctics[i] == 0)
     /*bypass*/
     place++;
   else
     {
         /*copy over data*/
         for(j=0; j<data_len; j++)
           set_bit(ds, ds_place++, get_bit(tdi_ds, place++));
     }
 delete_data_stream(tms_ds);
 delete_data_stream(tdi_ds);
}


void delay(int cycles)
{
 struct data_stream *tms_ds, *tdi_ds;


 /*create streams*/
 tms_ds = create_data_stream(cycles);
 tdi_ds = create_data_stream(cycles);

 board_send_bits(tdi_ds, tms_ds);

 delete_data_stream(tms_ds);
 delete_data_stream(tdi_ds);
 return;
}


void load_JTAGDR2_beg(struct data_stream *ds, int arctics[],
```

```
                    int data_length)
{
 struct data_stream *tms_ds, *tdi_ds;
 int place = 0;
 int ds_place = 0;
 int i, j;
 int len = 0;

 if(TEST>4)
  printf("load JTAGDR2_beg\n");
 if(logf_level>4)
  fprintf(logf, "load JTAGDR2_beg\n");

 /*figure out lenght of streams*/
 for(i=0; i<NA; i++)
  if(arctics[i] == 1)
   len++;

 len = len*data_length + (NA-len) + 5;

 /*create streams*/
 tms_ds = create_data_stream(len);
 tdi_ds = create_data_stream(len);


 /*get to shift DR stat from RTI*/
 set_bit(tms_ds, place++, ONE);
 set_bit(tms_ds, place++, ZERO);
 set_bit(tms_ds, place++, ZERO);

 /*shift DR*/
 for(i=0; i<NA; i++)
  if(arctics[i] == 0)
   /*bypass*/
   place++;
  else
   {
       /*copy in data*/
       for(j=0; j<data_length; j++)
         set_bit(tdi_ds, place++, get_bit(ds, ds_place++));
   }

 place--;
```

```
 /*go to Pause DR*/
 set_bit(tms_ds, place++, ONE);
 set_bit(tms_ds, place++, ZERO);

 board_send_bits(tdi_ds, tms_ds);

 /*bits returnd in tdi_ds, remove extra bits, and
   send back in ds,
   3 extra
   1 for each bypass
   data_length for each actual
   2 more extra*/

 place = 3;
 ds_place = 0;
 for(i=0; i<NA; i++)
   if(arctics[i] == 0)
     /*bypass*/
     place++;
   else
     {
         /*copy over data*/
         for(j=0; j<data_length; j++)
           set_bit(ds, ds_place++, get_bit(tdi_ds, place++));
     }

 delete_data_stream(tms_ds);
 delete_data_stream(tdi_ds);

}

void load_JTAGDR2(struct data_stream *ds, int arctics[],
                  int data_length)
{
 struct data_stream *tms_ds, *tdi_ds;
 int place = 0;
 int ds_place = 0;
 int i, j;
 int len = 0;

 if(TEST>4)
   printf("load JTAGDR2\n");
```

```
if(logf_level>4)
  fprintf(logf, "load JTAGDR2\n");

/*figure out length of streams*/
for(i=0; i<NA; i++)
  if(arctics[i] == 1)
    len++;

len = len*data_length + (NA-len) + 8;

/*create streams*/
tms_ds = create_data_stream(len);
tdi_ds = create_data_stream(len);


/*get to shift DR state from Pause-DR*/
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ZERO);     /* --> RTI */
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ZERO);
set_bit(tms_ds, place++, ZERO);

/*shift DR*/
for(i=0; i<NA; i++)
  if(arctics[i] == 0)
    /*bypass*/
    place++;
  else
    {
        /*copy in data*/
        for(j=0; j<data_length; j++)
          set_bit(tdi_ds, place++, get_bit(ds, ds_place++));
    }

place--;

/*go to Pause DR*/
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ZERO);

board_send_bits(tdi_ds, tms_ds);
```

```
 /*bits returnd in tdi_ds, remove extra bits, and
   send back in ds,
   6 extra
   1 for each bypass
   data_length for each actual
   2 more extra*/

 place = 6;
 ds_place = 0;
 for(i=0; i<NA; i++)
   if(arctics[i] == 0)
     /*bypass*/
     place++;
   else
     {
         /*copy over data*/
         for(j=0; j<data_length; j++)
           set_bit(ds, ds_place++, get_bit(tdi_ds, place++));
     }

 delete_data_stream(tms_ds);
 delete_data_stream(tdi_ds);
}

void load_JTAGDR2_end()
{
 struct data_stream *tms_ds, *tdi_ds;
 int place = 0;
 int ds_place = 0;
 int i;

 /*create streams*/
 tms_ds = create_data_stream(3);
 tdi_ds = create_data_stream(3);


 /*get to RTI stat from Pause-DR*/
 set_bit(tms_ds, place++, ONE);
 set_bit(tms_ds, place++, ONE);
 set_bit(tms_ds, place++, ZERO);

 board_send_bits(tdi_ds, tms_ds);
```

```
  delete_data_stream(tms_ds);
  delete_data_stream(tdi_ds);
}
```

D.14 jtag_trans2.c

```
/* This is the jtag_trans.c file for use with the Corelis board */

#include "ancis.h"

extern  FILE *logf;
extern  int logf_level;

void load_JTAGIR(enum JTAG_INST instr, enum JTAG_INST alternate,
                 int arctics[NA])
{
 struct data_stream *tdi_ds;
 int place = 0 ;
 int i, length;
 unsigned short *array_ptr;

 if(TEST>4)
  printf("      load JTAGIR instr: %x \n", instr);
 if(logf_level>4)
  fprintf(logf, "      load JTAGIR instr: %x \n", instr);

 /*length of tdi data stream: all jtag instructions
  are 4 bits long so length = NA*4*/

 length = NA*4;
 tdi_ds = create_data_stream(length);

 for(i=0; i<NA; i++)
  {
   if(arctics[i] == 0)

       /*put in alternate*/
       {
        set_bit(tdi_ds, place++, (alternate & 1));
        set_bit(tdi_ds, place++, ((alternate>>1) & 1));
        set_bit(tdi_ds, place++, ((alternate>>2) & 1));
        set_bit(tdi_ds, place++, ((alternate>>3) & 1));
       }
   else
       /*put in instr*/
       {
```

```
            set_bit(tdi_ds, place++, (instr & 1));
            set_bit(tdi_ds, place++, ((instr>>1) & 1));
            set_bit(tdi_ds, place++, ((instr>>2) & 1));
            set_bit(tdi_ds, place++, ((instr>>3) & 1));
          }
    }

  /* convert data stream to array for corelis driver */
  array_ptr = ds2array(tdi_ds);

  scan_ir(JTAG_BUS4, array_ptr, length, array_ptr);

  /* can also try
          scan_ir_turbo(JTAG_BUS4, array_ptr, length, array_ptr);
    but it is more complex and has not been tested */

  delete_data_stream(tdi_ds);

}

void load_JTAGDR(struct data_stream *ds, int arctics[NA],
                        int data_len)
{
  struct data_stream *tdi_ds;
  int place = 0;
  int ds_place = 0;
  int i, j;
  int len = 0;
  unsigned short *array_ptr;

  if(TEST>4)
    {
     printf("         load JTAGDR data:");
     print_stream(ds);
    }
  if(logf_level>4)
    {
     fprintf(logf, "         load JTAGDR data:");
     fprint_stream(ds);
    }

  /*figure out length of stream*/
  for(i=0; i<NA; i++)
```

```
 if(arctics[i] == 1)
  len++;

len = len*data_len + (NA-len);

/*create stream*/
tdi_ds = create_data_stream(len);

/*shift DR*/
for(i=0; i<NA; i++)
 if(arctics[i] == 0)
  /*bypass*/
  place++;
 else
  {
      /*copy in data*/
      for(j=0; j<data_len; j++)
        set_bit(tdi_ds, place++, get_bit(ds, ds_place++));
  }

/* convert data stream to array for corelis driver */
array_ptr = ds2array(tdi_ds);

scan_dr(JTAG_BUS4, array_ptr, len, array_ptr);

/* can also try
       scan_dr_turbo(JTAG_BUS4, array_ptr, len, array_ptr);
  but it is more complex and has not been tested */

/*bits returnd in tdi_ds, remove extra bits, and
 send back in ds,
 1 for each bypass
 data_len for each actual
*/

place = 0;
ds_place = 0;
for(i=0; i<NA; i++)
 if(arctics[i] == 0)
  /*bypass*/
  place++;
 else
  {
```

```
        /*copy over data*/
        for(j=0; j<data_len; j++)
          set_bit(ds, ds_place++, get_bit(tdi_ds, place++));
    }

 delete_data_stream(tdi_ds);
}

void load_JTAGDR2_beg(struct data_stream *ds, int arctics[],
                      int data_length)
{
 struct data_stream *tdi_ds;
 int place = 0;
 int ds_place = 0;
 int i, j;
 int len = 0;
 unsigned short *array_ptr;

 if(TEST>4)
  printf("load JTAGDR2_beg\n");
 if(logf_level>4)
  fprintf(logf, "load JTAGDR2_beg\n");

 /*figure out length of stream*/
 for(i=0; i<NA; i++)
  if(arctics[i] == 1)
    len++;

 len = len*data_length + (NA-len);

 /*create stream*/
 tdi_ds = create_data_stream(len);

 for(i=0; i<NA; i++)
  if(arctics[i] == 0)
    /*bypass*/
    place++;
  else
    {
        /*copy in data*/
        for(j=0; j<data_length; j++)
          set_bit(tdi_ds, place++, get_bit(ds, ds_place++));
    }
```

```
 /* convert data stream to array for corelis driver */
 array_ptr = ds2array(tdi_ds);

 scan_dr(JTAG_BUS4, array_ptr, len, array_ptr, BEG);

 /*bits returnd in tdi_ds, remove extra bits, and
  send back in ds,
  1 for each bypass
  data_length for each actual
 */

 place = 0;
 ds_place = 0;
 for(i=0; i<NA; i++)
  if(arctics[i] == 0)
   /*bypass*/
   place++;
  else
   {
       /*copy over data*/
       for(j=0; j<data_length; j++)
        set_bit(ds, ds_place++, get_bit(tdi_ds, place++));
   }

 delete_data_stream(tdi_ds);

}

void load_JTAGDR2(struct data_stream *ds, int arctics[],
                  int data_length)
{
 struct data_stream*tdi_ds;
 int place = 0;
 int ds_place = 0;
 int i, j;
 int len = 0;
 unsigned short *array_ptr;

 if(TEST>4)
  printf("load JTAGDR2\n");
 if(logf_level>4)
  fprintf(logf, "load JTAGDR2\n");
```

```
/*figure out length of stream*/
for(i=0; i<NA; i++)
 if(arctics[i] == 1)
  len++;

len = len*data_length + (NA-len);

/*create stream*/
tdi_ds = create_data_stream(len);

for(i=0; i<NA; i++)
 if(arctics[i] == 0)
  /*bypass*/
  place++;
 else
  {
      /*copy in data*/
      for(j=0; j<data_length; j++)
       set_bit(tdi_ds, place++, get_bit(ds, ds_place++));
  }

/* convert data stream to array for corelis driver */
array_ptr = ds2array(tdi_ds);

scan_dr(JTAG_BUS4, array_ptr, len, array_ptr, LOOP);

/*bits returnd in tdi_ds, remove extra bits, and
 send back in ds,
 1 for each bypass
 data_length for each actual
 */

place = 0;
ds_place = 0;
for(i=0; i<NA; i++)
 if(arctics[i] == 0)
  /*bypass*/
  place++;
 else
  {
      /*copy over data*/
      for(j=0; j<data_length; j++)
```

```
            set_bit(ds, ds_place++, get_bit(tdi_ds, place++));
        }

    delete_data_stream(tdi_ds);
}

void load_JTAGDR2_end()
{
    scan_dr2_end(JTAG_BUS4);
}
```

D.15 board1.h

```
/* This file is to be used when testing with the  Verilog model */

enum ports{TDI, TMS};
enum board_status{B_COMPLETED, B_IN_PROGRESS};

int board_send_bits(struct data_stream *tdi,
                    struct data_stream *tms);
/*
1.sends given chunk of bits out to the board (in one
piece.  bits must end in a pause state.  bits must
begin in the state the last instruction left them in.

2.sends a go instruction to the board.

3.waits till the board is finished, and reads the data
from TDO into the tdi data_stream.

***doesn't look at status***
*/

int board_reset();

/*resets the JTAG board

 sends/reads two data words to fill up the extra space
 in the fifo
*/


/*** lower level functions ***/

int board_send_stream(enum ports st, struct data_stream *ds);

/*sends stream to board.  fills in to an even
 number of bytes with the last bit in the stream.
 returns last bit of stream*/

int board_go(struct data_stream *tdo, enum bit last_bit_tms);
/*sends and captures data to tdo
 fills in and strips off byte of idle repeats of last_bit
```

```
 in both tms and tdi
 reads tdo
returns board to receiving state when done*/
```

D.16 board1.c

```
/* Temp board file - for testing with Verilog only */

#include "ancis.h"

extern  FILE *logf;       /* for log files */
extern  int logf_level;

int board_send_bits(struct data_stream *tdi,
                    struct data_stream *tms)
{

 int last_bit;


 /*check to make sure there are less then 4k-1 bytes of data */

 if(get_ds_length(tdi) > 3999*8)
  {
   printf("uh oh, attempt to send chunk bigger then the buffers \n");
   if (logf_level>0)
       fprintf(logf, "uh oh, attempt to send chunk bigger than the buffers \n");
   return(0);
  }

 /*fill buffers, send stream fills buffers to an even
  number of bits*/

 if (TEST>5)
  {
   printf("          Bits Sent To Board \n");
   printf("          TDI\n");
  }
 board_send_stream(TDI, tdi);

 if (TEST>5)
  printf("          TMS\n");

 last_bit = board_send_stream(TMS, tms);

 /*send a go signal: go sends bits, waits till done
```

```
  then returns whatever is sent back.  It also
  fills in and strips off the extra byte that fills
  the serial to parallel converter.*/
 /*returned bits are put back into tdi*/

 board_go(tdi, last_bit);

}

int board_send_stream(enum ports st, struct data_stream *ds)
{
 int i, j, k, len;
 int place = 0;
 unsigned short byte;
 struct data_stream *temp;

 /*send the main stream to the board*/
 /*length of stream to be sent*/
 len = get_ds_length(ds);

 /*send full bytes */
 for(i=0; i < (len/8);  i++)
   {
    get_byte(ds, place, &byte);
    if(st == TDI)
        board_write(TDI_ADDR, byte);
    else
        board_write(TMS_ADDR, byte);
    place = place + 8;
   }

    /* figure out what the last bit of the stream was */
    j = get_bit(ds, len);

/*if there wasn't an even number of bytes, pick up the last one*/
 if( (len % 8) > 0)
   {
    get_byte(ds, place, &byte);

    /* fill in extra bits of last byte with the last bit,
        then send it */

    if (j == 0)
```

```
        /*clear end of byte*/
        byte = byte & (255 >> (8-len%8));
    else
        /*fill in last bits with 1's */
        byte = byte | (255 << (len%8));

    if(st == TDI)
        board_write(TDI_ADDR, byte);
    else
        board_write(TMS_ADDR, byte);
   }

 return(j);
}

int board_reset()
{
 short s;

 board_write(COMMAND, RESET);

 return(1);
}

int board_go(struct data_stream *tdo,
             enum bit last_bit_tms)
{
 /*add one extra byte of info to end of the input streams,
   ignores the first  byte coming out*/

 int status;
 int place;
 int i;
 int ds_len;
 short s, j;

 /*fill in extra tdi byte*/
 if (TEST>5)
   printf("          TDI\n");
 board_write(TDI_ADDR, 0);

 /*fill in extra tms byte*/
 if (TEST>5)
```

```
  printf("          TMS\n");
 if (last_bit_tms == 0)
  {
   board_write(TMS_ADDR,0);
  }
 else
  {
   board_write(TMS_ADDR, 255);
  }

 /*tell board to go*/
 if (TEST>5)
  printf("          go command\n");
 board_write(COMMAND, START);

 /*wait for it to finish*/
 while(board_read(STATUS, &s))
        {
         if (s == DONE)
           break;
        }

 /*tell board to stop*/
 board_write(COMMAND, STOP);

 /*read data out*/

/* if the board was actually there and saving the last
  byte to send later you would ignore the first byte returned.
  as it we ignore the last byte instead
 board_read(TDO_ADDR, &s);
*/

 /*figure out data stream length rounding up to the
  nearest byte*/

 ds_len = get_ds_length(tdo);

 if ((ds_len % 8) == 0)
  ds_len = ds_len / 8;
 else
  ds_len = (ds_len / 8) +1;
```

```
 /*write rest into tdo*/
 for(i=0; i < ds_len; i++)
   {
    board_read(TDO_ADDR, &j);
    set_byte(tdo, (i*8), j);
   }

/*last byte ignored, for simulation only*/
 board_read(TDO_ADDR, &s);

}
```

D.17 board2.h

```
/* header file for use with the Corelis board */

/*much of this file is taken from test.c - the Corelis example software*/

/* Include files */
#include <dos.h>
#include <conio.h>
#include <string.h>


/* Boolean constants */
#define TRUE            1
#define FALSE           0

/* default base address of test bus controller board */

#define DEFAULT_TBC_BASE  0x0300

/* Here are the constants for the scan_dr2() function */

#define BEG             1     /* first time through */
#define LOOP            0     /* each time thereafter */

/* The following offsets from the base address are used to access the
   PSCs, registers and FIFOs on-board the PC-1149.1/100F */

#define PSC1_OFFSET     0x00
#define PSC2_OFFSET     0x08

#define JTAG_TDO_FIFO     0x10     /* data write - word/byte */
#define JTAG_TDI_FIFO     0x10     /* data read  - word/byte */
#define CONTROL1          0x14     /* data write - word */
#define STATUS1           0x14     /* data read  - word */
#define CONTROL2          0x16     /* data write - word */
#define STATUS2           0x16     /* data read  - word */
#define STATUS3           0x18     /* data read  - word */
#define PARALLEL_OUT      0x1A     /* data write - word */
#define PARALLEL_IN       0x1A     /* data read  - word */
#define PARALLEL_OUT_RDBK 0x1E     /* data read  - word */
```

```
/* PSC register offsets from start of PSC address */

#define TDO_BUFFER      0x00    /* data write */
#define COUNT_REG1      0x00    /* data read */
#define TDI_BUFFER      0x01    /* data read/write */
#define TMS0_BUFFER     0x02    /* data write */
#define COUNT_REG2      0x02    /* data read */
#define TMS1_BUFFER     0x03    /* data write */
#define COUNT_REG3      0x03    /* data read */
#define COUNTER_32      0x04    /* data write */
#define COUNT_REG0      0x04    /* data read */
#define MODE0           0x05    /* data read/write */
#define MODE1           0x06    /* data read/write */
#define MODE2           0x07    /* data read/write */

/* parameters used to specify JTAG bus for scan or self-test */

#define JTAG_BUS1       0       /* accessed through PSC1, TMS0 */
#define JTAG_BUS2       1       /* accessed through PSC1, TMS1 */
#define JTAG_BUS3       2       /* accessed through PSC2, TMS0 */
#define JTAG_BUS4       3       /* accessed through PSC2, TMS1 */

/* used in word-to-byte conversion routine */

#define LSB             0       /* used for getting LSB from word */
#define MSB             1       /* used for getting MSB from word */

/* test status */

#define PASS            0       /* test passes */
#define FAIL            1       /* test fails */
#define ABORT           2       /* test aborted */

/* types of loopback test (internal to PSC) */

#define LOOP_TDO        0       /* loopback of TDO */
#define LOOP_TMS0       1       /* loopback of TMS0 */
#define LOOP_TMS1       2       /* loopback of TMS1 */

#define SYNC_RESET      0x02    /* write to MODE2 to reset a PSC */

/* the following are used in the programming of the on-board clock
   generator */
```

```
#define CLK_12              0     /* dummy parameter - set clock 1/2 */
#define CLK_34              1     /* dummy parameter - set clock 3/4 */

#define NO_PRESCALE         0     /* clock, without /16 prescaler */
#define PRESCALE_16         1     /* clock, with /16 prescaler */

#define INT_CLK_30MHZ       0     /* use on-board 30MHz oscillator */
#define INT_CLK_40MHZ       1     /* use on-board 40MHz oscillator */
#define INT_CLK_50MHZ       2     /* use on-board 50MHz oscillator */
#define EXT_CLK             3     /* use external clock (TCKI) */

#define CLK_DIV_BY_2        0     /* divide selected clock by 2 */
#define CLK_DIV_BY_4        1     /* divide selected clock by 4 */
#define CLK_DIV_BY_6        2     /* divide selected clock by 6 */
#define CLK_DIV_BY_8        3     /* divide selected clock by 8 */
#define CLK_DIV_BY_10       4     /* divide selected clock by 10 */
#define CLK_DIV_BY_12       5     /* divide selected clock by 12 */
#define CLK_DIV_BY_14       6     /* divide selected clock by 14 */
#define CLK_DIV_BY_16       7     /* divide selected clock by 16 */
#define CLK_DIV_BY_18       8     /* divide selected clock by 18 */
#define CLK_DIV_BY_20       9     /* divide selected clock by 20 */
#define CLK_DIV_BY_22       10    /* divide selected clock by 22 */
#define CLK_DIV_BY_24       11    /* divide selected clock by 24 */
#define CLK_DIV_BY_26       12    /* divide selected clock by 26 */
#define CLK_DIV_BY_28       13    /* divide selected clock by 28 */
#define CLK_DIV_BY_30       14    /* divide selected clock by 30 */
#define CLK_DIV_BY_32       15    /* divide selected clock by 32 */

/* length of SCAN18374 registers */
#define ON_BOARD_IR_LEN     8
#define ON_BOARD_DR_LEN     42

/* SCAN18374 instruction codes */
#define EXTEST_18374        0
#define SAMPLE_18374        0x81
#define BYPASS_18374        0xFF

/* transmit and receive FIFO size (in words) */
#define TX_FIFO_SIZE        512
#define RX_FIFO_SIZE        512

/*-------------------------------------------------------------------*/
```

```
/* function protoypes */

int board_reset();
void scan_ir(unsigned short, unsigned short *, unsigned short, unsigned short *);
void scan_dr(unsigned short, unsigned short *, unsigned short, unsigned short *);
void scan_dr2(unsigned short, unsigned short *, unsigned short, unsigned short *,
unsigned short);
void scan_dr2_end(unsigned short);
void wait_for_ready(void);
unsigned short check_ready(void);
void write_psc(unsigned short, unsigned short, unsigned char);
void read_psc(unsigned short, unsigned short, unsigned char *);
void write_tdo(unsigned short, unsigned char);
void write_tms(unsigned short, unsigned short, unsigned char);
void read_tdi(unsigned short, unsigned char *);
void load_cnt_32(unsigned short, unsigned long);
unsigned long read_cnt_32(unsigned short);
unsigned char get_byte_word(unsigned short, unsigned char);
unsigned short build_word(unsigned char, unsigned char);
void pgrm_scan_clk(unsigned short, unsigned short, unsigned short, unsigned short);
void soft_reset(unsigned short);
void hard_reset(unsigned short);
void setbit(unsigned short *, unsigned short, unsigned short);
short getbit(unsigned short *, unsigned short);
void enable_jtag_selftest(unsigned short);
void disable_jtag_selftest(void);
void scan_ir_turbo(unsigned short, unsigned short *, unsigned short, unsigned short *);
void scan_dr_turbo(unsigned short, unsigned short *, unsigned short, unsigned short *);
unsigned short jtag_selftest(void);
unsigned short jtag_selftest_turbo(void);
```

D.18 board2.c

```
/* board file for use with Corelis Board */

/****************************************
 Much of this file is heavily based upon or taken directly from
 the Corelis example/test software (the file "test.c").

 See the file "board.doc" for more information on the functions
 contained in this file.
 ****************************************/

#include "ancis.h"

extern  FILE *logf;
extern  int logf_level;


/*--------------------------------------------------------------------*/

unsigned short tbc_base;              /* holds either default or user
                                      entered board base address */

/* save environment */
struct text_info dos_attributes;

/*--------------------------------------------------------------------*/



int board_reset()
{

 /* initialize base address */
 tbc_base = 0x300;

 hard_reset(TRUE);

 return(1);
}
```

```
/
**********************************************************************
**

    Function: scan_ir()

    Summary: Scans a bit stream into the TAP instruction register

      Usage: void scan_ir(test_bus, output, length, input)
          unsigned short test_bus;
          unsigned short *output;
          unsigned short length;
          unsigned short *input;

  Description: This routine scans an arbitrary length bit stream into a
          target TAP instruction register. This function has four
          parameters. The 'test_bus' parameter indicates which of
          the four JTAG busses will be used for the scan. The 'output'
          variable is an address to an array of byte to be scanned
          out the JTAG bus. The 'length' parameter specifies the
          length of the bitstream.  The 'input' variable is an
          address to an array of byte values where the data scanned
          in is to be stored.

      Notes: This function writes/reads data directly to/from the
          appropriate PSC. The on-board FIFOs are not used.

**********************************************************************
***/

void scan_ir(test_bus, output, length, input)
unsigned short test_bus;
unsigned short *output;
unsigned short length;
unsigned short *input;
{
  unsigned char cnt_status;          /* storage for status of 32-bit
                              counter */
  unsigned short numwords = 0;       /* number of whole words to send,
                              rounded up */
  unsigned short x;                  /* loop counter */
  unsigned char scan_in_lsb;         /* data scanned in, lsb */
  unsigned char scan_in_msb;         /* data scanned in, msb */
```

```
unsigned char shift_count;        /* holds right shift count in the
                                     event of odd bits */
unsigned short psc_offset;        /* holds offset of PSC, based on JTAG
                                     bus being used */
unsigned short tms_reg_offset;    /* offset of PSC TMS buffer/shifter,
                                     based on JTAG bus being used */

/* setup for appropriate JTAG bus */
if (test_bus == JTAG_BUS1) {
  psc_offset = PSC1_OFFSET;
  tms_reg_offset = TMS0_BUFFER;
}
else if (test_bus == JTAG_BUS2) {
  psc_offset = PSC1_OFFSET;
  tms_reg_offset = TMS1_BUFFER;
}
else if (test_bus == JTAG_BUS3) {
  psc_offset = PSC2_OFFSET;
  tms_reg_offset = TMS0_BUFFER;
}
else if (test_bus == JTAG_BUS4) {
  psc_offset = PSC2_OFFSET;
  tms_reg_offset = TMS1_BUFFER;
}

/* compute number of words to load into TDO shifter/buffer. get number
  of whole words and add 1 to it if it does not fall on even word
  boundary */
numwords = length / 16;
if ((length % 16) != 0)
  ++numwords;

/* if length does not align to byte boundary, we have to shift right
  to move odd bits received into lsb's */
shift_count = (8 - (length % 8)) % 8;

/* remove TCK dependency on TDI and TDO buffers. restore dependency on
  TMS buffer */
if (tms_reg_offset == TMS0_BUFFER) {
  write_psc(psc_offset, MODE0, 0x30);
}
else {
  write_psc(psc_offset, MODE0, 0x28);
```

```
}

/* move TAP controller into SHIFT-IR state */
load_cnt_32(psc_offset, 5);
write_tms(psc_offset, tms_reg_offset, 0x06);

/* wait for 32-bit counter to reach terminal count */
do {
  read_psc(psc_offset, MODE2, &cnt_status);
  cnt_status = cnt_status & 0x20;
} while (cnt_status == 0);

/* restore TCK dependency on TDI and TDO buffers. remove dependency on
  TMS. set TMS high mode */
write_psc(psc_offset, MODE0, 0xE2);

/* load 32-bit counter with number of bits to scan */
load_cnt_32(psc_offset, (unsigned long) length);

/* scan out all of the whole words */
for (x = 0; x < numwords-1; x++) {
  write_tdo(psc_offset, get_byte_word(output[x], LSB));
  read_tdi(psc_offset, &scan_in_lsb);
  write_tdo(psc_offset, get_byte_word(output[x], MSB));
  read_tdi(psc_offset, &scan_in_msb);
  input[x] = build_word(scan_in_msb, scan_in_lsb);
}

/* scan out the last word */
if ((((length - 1) / 8) % 2) == 0) {
  write_tdo(psc_offset, get_byte_word(output[x], LSB));
  read_tdi(psc_offset, &scan_in_lsb);
  scan_in_lsb = scan_in_lsb >> shift_count;
  input[x] = build_word(0, scan_in_lsb);
}
else {
  write_tdo(psc_offset, get_byte_word(output[x], LSB));
  read_tdi(psc_offset, &scan_in_lsb);
  write_tdo(psc_offset, get_byte_word(output[x], MSB));
  read_tdi(psc_offset, &scan_in_msb);
  scan_in_msb = scan_in_msb >> shift_count;
  input[x] = build_word(scan_in_msb, scan_in_lsb);
}
```

```
  /* restore TCK dependency on TMS buffer. remove TCK dependency on TDI
    and TDO buffers. */
  if (tms_reg_offset == TMS0_BUFFER) {
    write_psc(psc_offset, MODE0, 0x30);
  }
  else {
    write_psc(psc_offset, MODE0, 0x28);
  }

  /* Apparently the parallel to serial converter on the board sends
    a one on the TMS line after the data is sent to get the TAP
    Controller out of the Shift-IR state. So at this point we should
    be in the Exit1-IR state */

  /* load TMS buffer to move TAP controller to RUN-TEST/IDLE state */
  load_cnt_32(psc_offset, 2);
  write_tms(psc_offset, tms_reg_offset, 0x01);

  /* wait for 32-bit counter to reach terminal count */
  do  {
    read_psc(psc_offset, MODE2, &cnt_status);
    cnt_status = cnt_status & 0x20;
  }  while (cnt_status == 0);

} /* end scan_ir() */


/
**********************************************************************
**

     Function: scan_dr()

      Summary: Scans a bit stream out the TAP data path

        Usage: void scan_dr(test_bus, output, length, input)
             unsigned short test_bus;
             unsigned short *output;
             unsigned short length;
             unsigned short *input;

   Description: This routine scans an arbitrary length bit stream into a
```

```
        target TAP data register. This function has four parameters.
        The 'test_bus' parameter indicates which of the four JTAG
        busses will be used for the scan. The 'output' variable is
        an address to an array of byte to be scanned out the JTAG
        bus. The 'length' parameter specifies the length of the
        bitstream. The 'input' variable is an address to an array
        of byte values where the data scanned in is to be stored.

    Notes: This function writes/reads data directly to/from the
        appropriate PSC. The on-board FIFOs are not used.

*************************************************************************
***/

void scan_dr(test_bus, output, length, input)
unsigned short test_bus;
unsigned short *output;
unsigned short length;
unsigned short *input;
{
  unsigned char cnt_status;          /* storage for status of 32-bit
                                      counter */
  unsigned short numwords = 0;       /* number of whole words to send,
                                      rounded up */
  unsigned short x;                  /* loop counter */
  unsigned char scan_in_lsb;         /* data scanned in, lsb */
  unsigned char scan_in_msb;         /* data scanned in, msb */
  unsigned char shift_count;         /* holds right shift count in the
                                      event of odd bits */
  unsigned short psc_offset;         /* holds offset of PSC, based on JTAG
                                      bus being used */
  unsigned short tms_reg_offset;     /* offset of PSC TMS buffer/shifter,
                                      based on JTAG bus being used */


  /* setup for appropriate JTAG bus */
  if (test_bus == JTAG_BUS1) {
    psc_offset = PSC1_OFFSET;
    tms_reg_offset = TMS0_BUFFER;
  }
  else if (test_bus == JTAG_BUS2) {
    psc_offset = PSC1_OFFSET;
    tms_reg_offset = TMS1_BUFFER;
```

```
}
else if (test_bus == JTAG_BUS3) {
  psc_offset = PSC2_OFFSET;
  tms_reg_offset = TMS0_BUFFER;
}
else if (test_bus == JTAG_BUS4) {
  psc_offset = PSC2_OFFSET;
  tms_reg_offset = TMS1_BUFFER;
}

/* compute number of words to load into TDO shifter/buffer. get number
  of whole words and add 1 to it if it does not fall on even word
  boundary */
numwords = length / 16;
if ((length % 16) != 0)
  ++numwords;

/* if length does not align to byte boundary, we have to shift right
  to move odd bits received into lsb's */
shift_count = (8 - (length % 8)) % 8;

/* remove TCK dependency on TDI and TDO buffers. restore dependency on
  TMS buffer */
if (tms_reg_offset == TMS0_BUFFER) {
  write_psc(psc_offset, MODE0, 0x30);
}
else {
  write_psc(psc_offset, MODE0, 0x28);
}

/* move TAP controller into SHIFT-DR state */
load_cnt_32(psc_offset, 4);
write_tms(psc_offset, tms_reg_offset, 0x02);

/* wait for 32-bit counter to reach terminal count */
do {
  read_psc(psc_offset, MODE2, &cnt_status);
  cnt_status = cnt_status & 0x20;
} while (cnt_status == 0);

/* restore TCK dependency on TDI and TDO buffers. remove dependency on
  TMS. set TMS high mode */
write_psc(psc_offset, MODE0, 0xE2);
```

```
/* load 32-bit counter with number of bits to scan */
load_cnt_32(psc_offset, (unsigned long) length);

/* scan out all of the whole words */
for (x = 0; x < numwords-1; x++) {
  write_tdo(psc_offset, get_byte_word(output[x], LSB));
  read_tdi(psc_offset, &scan_in_lsb);
  write_tdo(psc_offset, get_byte_word(output[x], MSB));
  read_tdi(psc_offset, &scan_in_msb);
  input[x] = build_word(scan_in_msb, scan_in_lsb);
}

/* scan out the last word */
if ((((length - 1) / 8) % 2) == 0) {
  write_tdo(psc_offset, get_byte_word(output[x], LSB));
  read_tdi(psc_offset, &scan_in_lsb);
  scan_in_lsb = scan_in_lsb >> shift_count;
  input[x] = build_word(0, scan_in_lsb);
}
else {
  write_tdo(psc_offset, get_byte_word(output[x], LSB));
  read_tdi(psc_offset, &scan_in_lsb);
  write_tdo(psc_offset, get_byte_word(output[x], MSB));
  read_tdi(psc_offset, &scan_in_msb);
  scan_in_msb = scan_in_msb >> shift_count;
  input[x] = build_word(scan_in_msb, scan_in_lsb);
}

/* restore TCK dependency on TMS buffer. remove TCK dependency on TDI
  and TDO buffers. */
if (tms_reg_offset == TMS0_BUFFER) {
  write_psc(psc_offset, MODE0, 0x30);
}
else {
  write_psc(psc_offset, MODE0, 0x28);
}

/* Apparently the parallel to serial converter on the board sends
  a one on the TMS line after the data is sent to get the TAP
  Controller out of the Shift-DR state. So at this point we should
  be in the Exit1-DR state */
```

```
  /* load TMS buffer to move TAP controller to RUN-TEST/IDLE state */
  load_cnt_32(psc_offset, 2);
  write_tms(psc_offset, tms_reg_offset, 0x01);

  /* wait for 32-bit counter to reach terminal count */
  do {
    read_psc(psc_offset, MODE2, &cnt_status);
    cnt_status = cnt_status & 0x20;
  } while (cnt_status == 0);
} /* end scan_dr() */


/
**********************************************************************
**

    Function: scan_dr2()

This function is based on scan_dr() above and is still very similar. The
only difference is in the TAP states that are entered. This function also
takes one extra parameter, unsigned short begin. This variable is a flag
indicating whether or not this is the first time the function is entered
during one test. If begin is equal to BEG, this is the beginning of the
test, and the TAP Controller should be in the RTI state. Otherwise the
TAP Controller should be in the Pause-DR state. In either case, the TAP
Controller is left in the Pause-DR state when the function returns.

**********************************************************************
***/

void scan_dr2(test_bus, output, length, input, begin)
unsigned short test_bus;
unsigned short *output;
unsigned short length;
unsigned short *input;
unsigned short begin;
{
  unsigned char cnt_status;         /* storage for status of 32-bit
                                    counter */
  unsigned short numwords = 0;      /* number of whole words to send,
                                    rounded up */
  unsigned short x;                 /* loop counter */
  unsigned char scan_in_lsb;        /* data scanned in, lsb */
```

```
unsigned char scan_in_msb;        /* data scanned in, msb */
unsigned char shift_count;        /* holds right shift count in the
                                     event of odd bits */
unsigned short psc_offset;        /* holds offset of PSC, based on JTAG
                                     bus being used */
unsigned short tms_reg_offset;    /* offset of PSC TMS buffer/shifter,
                                     based on JTAG bus being used */


/* setup for appropriate JTAG bus */
if (test_bus == JTAG_BUS1) {
  psc_offset = PSC1_OFFSET;
  tms_reg_offset = TMS0_BUFFER;
}
else if (test_bus == JTAG_BUS2) {
  psc_offset = PSC1_OFFSET;
  tms_reg_offset = TMS1_BUFFER;
}
else if (test_bus == JTAG_BUS3) {
  psc_offset = PSC2_OFFSET;
  tms_reg_offset = TMS0_BUFFER;
}
else if (test_bus == JTAG_BUS4) {
  psc_offset = PSC2_OFFSET;
  tms_reg_offset = TMS1_BUFFER;
}

/* compute number of words to load into TDO shifter/buffer. get number
  of whole words and add 1 to it if it does not fall on even word
  boundary */
numwords = length / 16;
if ((length % 16) != 0)
  ++numwords;

/* if length does not align to byte boundary, we have to shift right
  to move odd bits received into lsb's */
shift_count = (8 - (length % 8)) % 8;

/* remove TCK dependency on TDI and TDO buffers. restore dependency on
  TMS buffer */
if (tms_reg_offset == TMS0_BUFFER) {
  write_psc(psc_offset, MODE0, 0x30);
}
```

```
else {
  write_psc(psc_offset, MODE0, 0x28);
}

/* move TAP controller into SHIFT-DR state */
/* Here's where this function differs from scan_dr() */

if (begin == BEG) {
 /* We are starting from RTI */
 load_cnt_32(psc_offset, 4);
 write_tms(psc_offset, tms_reg_offset, 0x02);
}
else {
 /* We are starting in the Pause-DR state */
 load_cnt_32(psc_offset, 6);
 write_tms(psc_offset, tms_reg_offset, 0x0b);
}

/* wait for 32-bit counter to reach terminal count */
do  {
  read_psc(psc_offset, MODE2, &cnt_status);
  cnt_status = cnt_status & 0x20;
}  while (cnt_status == 0);

/* restore TCK dependency on TDI and TDO buffers. remove dependency on
  TMS. set TMS high mode */
write_psc(psc_offset, MODE0, 0xE2);

/* load 32-bit counter with number of bits to scan */
load_cnt_32(psc_offset, (unsigned long) length);

/* scan out all of the whole words */
for (x = 0; x < numwords-1; x++) {
  write_tdo(psc_offset, get_byte_word(output[x], LSB));
  read_tdi(psc_offset, &scan_in_lsb);
  write_tdo(psc_offset, get_byte_word(output[x], MSB));
  read_tdi(psc_offset, &scan_in_msb);
  input[x] = build_word(scan_in_msb, scan_in_lsb);
}

/* scan out the last word */
if ((((length - 1) / 8) % 2) == 0) {
  write_tdo(psc_offset, get_byte_word(output[x], LSB));
```

```
    read_tdi(psc_offset, &scan_in_lsb);
    scan_in_lsb = scan_in_lsb >> shift_count;
    input[x] = build_word(0, scan_in_lsb);
  }
  else {
    write_tdo(psc_offset, get_byte_word(output[x], LSB));
    read_tdi(psc_offset, &scan_in_lsb);
    write_tdo(psc_offset, get_byte_word(output[x], MSB));
    read_tdi(psc_offset, &scan_in_msb);
    scan_in_msb = scan_in_msb >> shift_count;
    input[x] = build_word(scan_in_msb, scan_in_lsb);
  }

  /* restore TCK dependency on TMS buffer. remove TCK dependency on TDI
    and TDO buffers. */
  if (tms_reg_offset == TMS0_BUFFER) {
    write_psc(psc_offset, MODE0, 0x30);
  }
  else {
    write_psc(psc_offset, MODE0, 0x28);
  }

  /* Apparently the parallel to serial converter on the board sends
    a one on the TMS line after the data is sent to get the TAP
    Controller out of the Shift-DR state. So at this point we should
    be in the Exit1-DR state */

  /* load TMS buffer to move TAP controller to Pause-DR state */
  load_cnt_32(psc_offset, 1);
  write_tms(psc_offset, tms_reg_offset, 0x00);

  /* wait for 32-bit counter to reach terminal count */
  do  {
    read_psc(psc_offset, MODE2, &cnt_status);
    cnt_status = cnt_status & 0x20;
  }  while (cnt_status == 0);
} /* end scan_dr2() */


/
**********************************************************************
**
```

Function: scan_dr2_end()

This function just moves the TAP Controller from the Pause-DR state to the Run-Test/Idle (RTI) state. This is used after scan_dr2() has been used leaving the TAP Controller in the Pause-DR state.

```
**********************************************************************
***/

void scan_dr2_end(test_bus)
unsigned short test_bus;
{
  unsigned char cnt_status;          /* storage for status of 32-bit
                                     counter */
  unsigned short psc_offset;         /* holds offset of PSC, based on JTAG
                                     bus being used */
  unsigned short tms_reg_offset;     /* offset of PSC TMS buffer/shifter,
                                     based on JTAG bus being used */


  /* setup for appropriate JTAG bus */
  if (test_bus == JTAG_BUS1) {
    psc_offset = PSC1_OFFSET;
    tms_reg_offset = TMS0_BUFFER;
  }
  else if (test_bus == JTAG_BUS2) {
    psc_offset = PSC1_OFFSET;
    tms_reg_offset = TMS1_BUFFER;
  }
  else if (test_bus == JTAG_BUS3) {
    psc_offset = PSC2_OFFSET;
    tms_reg_offset = TMS0_BUFFER;
  }
  else if (test_bus == JTAG_BUS4) {
    psc_offset = PSC2_OFFSET;
    tms_reg_offset = TMS1_BUFFER;
  }

  /* restore TCK dependency on TMS buffer. remove TCK dependency on TDI
    and TDO buffers. */
  if (tms_reg_offset == TMS0_BUFFER) {
    write_psc(psc_offset, MODE0, 0x30);
  }
```

```
  else {
    write_psc(psc_offset, MODE0, 0x28);
  }

  /* load TMS buffer to move TAP controller to RUN-TEST/IDLE state */
  load_cnt_32(psc_offset, 3);
  write_tms(psc_offset, tms_reg_offset, 0x03);

  /* wait for 32-bit counter to reach terminal count */
  do {
    read_psc(psc_offset, MODE2, &cnt_status);
    cnt_status = cnt_status & 0x20;
  } while (cnt_status == 0);
} /* end scan_dr2_end() */


/
**********************************************************************
**

     Function: wait_for_ready()

     Summary: Returns when SCANPSC100F is ready to be accessed.

      Usage: void wait_for_ready()

  Description: This procedure polls the XRDY bit in Status Register 2
               until the PSCs on-board are ready to be accessed. No
               parameters are required.

       Notes: The XRDY bit in the Status Register 2 is polled until
              a low level is observed, indicating that the PSCs are
              ready to be accessed. The XRDY bit in Status Register 2
              is an OR function of the 2 PSCs (i.e., if either of
              the PSCs has been accessed and is not ready, this bit
              will be set to a high).

**********************************************************************
***/

void wait_for_ready()
{
```

```
  /* wait for XRDY signal before accessing board */
  while((inport(STATUS2+tbc_base) & 0x2000) != 0x0000);

} /* end wait_for_ready() */


/
*************************************************************************
**

     Function: check_ready()

      Summary: Returns a value of TRUE when parallel/serial converters
               are ready to be accessed.

        Usage: unsigned short check_ready()

   Description: This procedure checks the XRDY* bit in Status Register 2
               to determine if the parallel/serial converters are
               ready to be accessed. If XRDY* is low, the parallel/serial
               converters are ready to be accessed and a value of TRUE
               is returned. Otherwise, a value of FALSE is returned.

*************************************************************************
***/

unsigned short check_ready()
{
 /* check XRDY*. if XRDY* is low, return TRUE. otherwise, return
    FALSE */
 if ((inport(STATUS2+tbc_base) & 0x2000) != 0x0000) {
    return FALSE;
 }
 else {
    return TRUE;
 }

}
```

```
/
**********************************************************************
**

     Function: write_psc()

      Summary: Writes a value to a register on one of the PSCs.

        Usage: void write_psc(psc_offset, psc_reg_offset, data)
               unsigned short psc_offset, psc_reg_offset;
               unsigned char data;

   Description: This routine first checks to make sure that the on-board
                PSCs are ready to be accessed. When the PSCs are ready to
                be accessed, the specified data is written to the register.
                The parameter 'psc_offset' specifes the offset, from the
                board base address, of the PSC to which data is being
                written. The parameter 'register_offset' specifies offset,
                internal to a PSC, of the register to which data is being
                written. The 'data' parameter holds the data to be written.

     Variables: 'tbc_base' must be set to the base address of the test
                controller board before using this function.

**********************************************************************
***/

void write_psc(psc_offset, psc_reg_offset, data)
unsigned short psc_offset, psc_reg_offset;
unsigned char data;
{

  wait_for_ready();
  outportb(tbc_base+psc_offset+psc_reg_offset, data);

} /* end write_psc() */


/
**********************************************************************
**

     Function: read_psc()
```

```
    Summary: Reads the value of one of the registers in one of the PSCs.

     Usage: void read_psc(psc_offset, psc_reg_offset, data)
          unsigned short psc_offset, psc_reg_offset;
          unsigned char * data;

 Description: This routine first checks to make sure that the on-board
          PSCs are ready to be accessed. When the PSCs are ready to
          be accessed, the specified data is read from the register.
          The parameter 'psc_offset' specifes the offset, from the
          board base address, of the PSC from which data is being
          read. The parameter 'register_offset' specifies offset,
          internal to a PSC, of the register from which data is being
          read. The 'data' parameter contains the address where the
          read data is to be stored.


*************************************************************************
***/

void read_psc(psc_offset, psc_reg_offset, data)
unsigned short psc_offset, psc_reg_offset;
unsigned char *data;
{

  wait_for_ready();
  *data = inportb(tbc_base+psc_offset+psc_reg_offset);

} /* end read_psc() */


/
*************************************************************************
**

    Function: write_tdo()

    Summary: Writes data to the TDO shifter/buffer on a PSC.

     Usage: void write_tdo(psc_offset, data)
          unsigned short psc_offset;
          unsigned char data;
```

```
   Description: This routine polls the MODE2 register of the PSC until the
                TDO Status is set (TDO not full).  At that time, it writes
                the specified data to the TDO shifter/buffer. This function
                has two parameters. The parameter 'psc_offset' specifes the
                offset, from the board base address, of the PSC to which
                data is being written. The 'data' parameter specifies the
                data to be written to the TDO buffer.

*************************************************************************
***/

void write_tdo(psc_offset, data)
unsigned short psc_offset;
unsigned char data;
{
   unsigned char mode2_data;          /* holds MODE2 data from PSC */

   /* loop while TDO buffer is full */
   do {
     read_psc(psc_offset, MODE2, &mode2_data);
   } while ((mode2_data & 0x80) == 0);

   /* TDO buffer has room, so write the data */
   write_psc(psc_offset, TDO_BUFFER, data);

} /* end write_tdo() */


/
*************************************************************************
**

     Function: write_tms()

     Summary: Writes data to the TMS shifter/buffer on a PSC.

       Usage: void write_tms(psc_offset, tms_reg_offset, data)
              unsigned short psc_offset, tms_reg_offset;
              unsigned char data;

   Description: This routine polls the MODE2 register of the PSC until the
                TMS0/1 Status is set (TMS0/1 not full).  At that time, it
```

```
        writes the specified data to the TMS0/1 shifter/buffer. This
        function has three parameters. The parameter 'psc_offset'
        specifes the offset, from the board base address, of the PSC
        to which data is being written. The 'tms_reg_offset'
        parameter specifes the offset of either the TMS0 or TMS1
        buffer/shifter. The parameter 'data' specifies the data to
        be written to the TMS0/1 shifter/buffer.

**********************************************************************
***/

void write_tms(psc_offset, tms_reg_offset, data)
unsigned short psc_offset, tms_reg_offset;
unsigned char data;
{
  unsigned char mode2_data;          /* holds MODE2 data from PSC */
  unsigned char tms_stat_mask;       /* mask to get TMS status from
                                        MODE2 data */


  /* configure for the appropriate TMS */
  if (tms_reg_offset == TMS0_BUFFER) {
    tms_stat_mask = 0x10;
  }
  else {
    tms_stat_mask = 0x08;
  }

  /* loop while TMS buffer/shifter is full */
  do  {
    read_psc(psc_offset, MODE2, &mode2_data);
  }  while ((mode2_data & tms_stat_mask) == 0);

  /* TMS buffer/shifter has room, so write the data */
  write_psc(psc_offset, tms_reg_offset, data);

} /* end write_tms() */


/
**********************************************************************
**
```

```
    Function: read_tdi()

     Summary: Reads data from the TDI shifter/buffer on a PSC.

      Usage: void read_tdi(psc_offset, data)
           unsigned short psc_offset;
           unsigned char *data;

  Description: This routine polls the MODE2 register of the PSC until the
           TDI Status is set (TDI not empty).  At that time, it reads
           a single byte from the TDI buffer/shifter. This function
           has two parameters. The parameter 'psc_offset' specifes
           the offset, from the board base address, of the PSC from
           which data is being read. The 'data' parameter specifies
           a pointer to storage for data read from TDI.

*************************************************************************
***/

void read_tdi(psc_offset, data)
unsigned short psc_offset;
unsigned char *data;
{

  unsigned char mode2_data;          /* holds MODE2 data from PSC */

  /* loop while TDI buffer is empty */
  do {
    read_psc(psc_offset, MODE2, &mode2_data);
  } while ((mode2_data & 0x40) == 0);


  /* TDI buffer has at least one byte of data, so do read */
  read_psc(psc_offset, TDI_BUFFER, data);

} /* end read_tdi() */


/
*************************************************************************
**

    Function: load_cnt_32()
```

```
   Summary: Loads data into the 32-bit counter of a PSC.

     Usage: void load_cnt_32(psc_offset, data)
            unsigned short psc_offset;
            unsigned long data;

 Description: This procedure takes the specified data and loads it into
            the 32-bit counter of a PSC. This function has two
            parameters. The parameter 'psc_offset' specifes the offset,
            from the board base address, of the PSC in which the counter
            is being loaded. The parameter 'data' specifies the data
            to be loaded into the counter.

*************************************************************************
***/

void load_cnt_32(psc_offset, data)
unsigned short psc_offset;
unsigned long data;
{
  unsigned char cnt_data;              /* storage to hold bytes
                                       of data sent to 32-bit
                                       counter */

  /* grab LSB of counter data */
  cnt_data = data;

  /* load LSB of data into counter */
  write_psc(psc_offset, COUNTER_32, cnt_data);

  /* shift counter data and grab second byte */
  data = data >> 8;
  cnt_data = data;

  /* load second byte of data into counter */
  write_psc(psc_offset, COUNTER_32, cnt_data);

  /* shift counter data and grab third byte */
  data = data >> 8;
  cnt_data = data;

  /* load third byte of data into counter */
```

```
   write_psc(psc_offset, COUNTER_32, cnt_data);

   /* shift counter data and grab MSB */
   data = data >> 8;
   cnt_data = data;

   /* load MSB of data into counter */
   write_psc(psc_offset, COUNTER_32, cnt_data);

} /* end load_cnt_32() */


/
**********************************************************************
**

     Function: read_cnt_32()

      Summary: Reads data from the 32-bit counter of a PSC.

        Usage: unsigned long read_cnt_32(psc_offset)
               unsigned short psc_offset;

   Description: This procedure reads data from the four counter registers
                in a PSC and combines them into a long, which is returned
                to the calling function. This function has two parameters.
                The parameter 'psc_offset' specifes the offset, from the
                board base address, of the PSC in which the counter is being
                loaded.

**********************************************************************
***/

unsigned long read_cnt_32(psc_offset)
unsigned short psc_offset;
{
   unsigned long cnt_data = 0;
   unsigned char cnt_reg_data;


   /* read Counter Register 3 to get MSB of counter data */
   read_psc(psc_offset, COUNT_REG3, &cnt_reg_data);
   cnt_data = cnt_data | ((unsigned long) cnt_reg_data);
```

```
    cnt_data = cnt_data << 8;

    /* read Counter Register 2 to get third byte of counter data */
    read_psc(psc_offset, COUNT_REG2, &cnt_reg_data);
    cnt_data = cnt_data | ((unsigned long) cnt_reg_data);
    cnt_data = cnt_data << 8;

    /* read Counter Register 1 to get second byte of counter data */
    read_psc(psc_offset, COUNT_REG1, &cnt_reg_data);
    cnt_data = cnt_data | ((unsigned long) cnt_reg_data);
    cnt_data = cnt_data << 8;

    /* read Counter Register 0 to get LSB of counter data */
    read_psc(psc_offset, COUNT_REG0, &cnt_reg_data);
    cnt_data = cnt_data | ((unsigned long) cnt_reg_data);

    return(cnt_data);

} /* end read_cnt_32() */


/
*************************************************************************
**

      Function: get_byte_word()

      Summary: Procedure to get byte data from word.

        Usage: unsigned char get_byte_word(data)
               unsigned short data;
               unsigned char byte;

    Description: This procedure gets the MSB or LSB from the word passed
               to it as a parameter.

*************************************************************************
***/

unsigned char get_byte_word(data, byte)
unsigned short data;
unsigned char byte;
{
```

```
  unsigned char data_byte;

  if (byte == LSB) {
    data_byte = data;
  }

  else {
    data_byte = (data >> 8);
  }

  /* return data */
  return data_byte;

} /* end get_byte_word() */


/
**********************************************************************
**

     Function: build_word()

     Summary: Procedure to combine two bytes into a word.

       Usage: unsigned short build_word(msb_data, lsb_data)
            unsigned char msb_data;
            unsigned char lsb_data;

   Description: This procedure combines two bytes into a word.

**********************************************************************
***/

unsigned short build_word(msb_data, lsb_data)
unsigned char msb_data;
unsigned char lsb_data;
{
  unsigned short data_word;

  data_word = (unsigned short) msb_data;
  data_word = data_word << 8;
  data_word = data_word | ((unsigned short) lsb_data);
```

```
  /* return data */
  return data_word;

} /* end build_word() */


/
**********************************************************************
**

     Function: pgrm_scan_clk()

      Summary: Procedure used to program the on-board clock generator.

        Usage: void pgrm_scan_clk(clock, clk_select, prescaler, clk_divider)
               unsigned short clock;
               unsigned short clk_select;
               unsigned short prescaler;
               unsigned short clk_divider;

   Description: This function can be used to program the clock generator
               for either JTAG channels 1/2 or JTAG channels 3/4. This
               function requires four parameters. The 'clock' parameter
               is used to select the clock for either JTAG busses 1/2
               or JTAG busses 3/4 for programming. The 'clk_select'
               parameter is used to select a clock source from one of
               the internal oscillators or from the external connector.
               The 'prescaler' parameter is used to select between '/16
               prescaling' or 'no prescaling' to be applied to the clock
               selected. The 'clk_divider' parameter is used to select
               the divide ratio applied to the clock selected.

**********************************************************************
***/

void pgrm_scan_clk(clock, clk_select, prescaler, clk_divider)
unsigned short clock;
unsigned short clk_select;
unsigned short prescaler;
unsigned short clk_divider;
{
  unsigned short control1_data;
```

```
/* get current clock configuration */
control1_data = inport(STATUS1+tbc_base);

if (clock == CLK_12) {

  /* clear bits for JTAG 1/2 */
  control1_data = control1_data & 0xFF00;

  /* OR in clock select */
  control1_data = control1_data | (clk_select << 5);

  /* OR in prescaler */
  control1_data = control1_data | (prescaler << 4);

  /* OR in divide ratio */
  control1_data = control1_data | clk_divider;

}
else if (clock == CLK_34) {

  /* clear bits for JTAG 3/4 */
  control1_data = control1_data & 0x00FF;

  /* OR in clock select */
  control1_data = control1_data | (clk_select << 13);

  /* OR in prescaler */
  control1_data = control1_data | (prescaler << 12);

  /* OR in divide ratio */
  control1_data = control1_data | (clk_divider << 8);

}
else {
  textcolor(LIGHTRED);
  cputs("Invalid clock selected for modification. \r\n");
  textcolor(WHITE);
}

outport(CONTROL1+tbc_base, control1_data);
```

```
} /* end pgrm_scan_clk() */


/
**********************************************************************
**

     Function: soft_reset()

      Summary: Provides a software reset of a PSC.

        Usage: void soft_reset(psc_offset)
               unsigned short psc_offset;

   Description: This function performs a soft reset of a PSC by writing to
                the MODE2 register. The parameter 'psc_offset' is used to
                indicate which PSC to reset.

        Notes: According to the data sheet for the SCANPSC100F, the RESET
               bit in the MODE2 register will return to zero when the
               reset operation is complete.


**********************************************************************
***/

void soft_reset(psc_offset)
unsigned short psc_offset;
{

  /* wait for SCANPSC100F to finish with previous access */
  wait_for_ready();

  /* set the RESET bit in the MODE 2 register of appropriate converter */
  write_psc(psc_offset, MODE2, SYNC_RESET);

} /* end soft_reset() */


/
**********************************************************************
**
```

```
    Function: hard_reset()

     Summary: Provides a hardware reset of the test controller board
              and targets.

       Usage: void hard_reset()

 Description: Write data to test controller board to perform hardware
              reset of all board functions and targets (TRST*). Parameter
              'rs422_enable' is used to indicate whether JTAG channel 4
              is RS-422 or standard interface. If 'rs422_enable' is set
              to TRUE, the RS-422 drivers and receivers will be enabled
              upon return.

       Notes: Following the execution of this procedure, both PSCs should
              be reset (refer to data sheet for SCANPSC100F for more
              information) and all targets should be reset (TRST* for all
              JTAG channels asserted). JTAG clocks will be set to their
              lowest internal frequency, parallel outputs will be enabled,
              self-test will be disabled, and PSC outputs will be enabled.


**********************************************************************
***/

void hard_reset(rs422_enable)
unsigned short rs422_enable;
{

  /* MODIFIED by cwward - clocks set to 10MHz */
  pgrm_scan_clk(CLK_12, INT_CLK_40MHZ, NO_PRESCALE, CLK_DIV_BY_4);
  pgrm_scan_clk(CLK_34, INT_CLK_40MHZ, NO_PRESCALE, CLK_DIV_BY_4);

  /* delay after switching clocks */
  delay(1);

  /* write to Control Register 2 to perform reset of PSCs, FIFOs, and
    targets (with or without RS-422) */
  if (rs422_enable == TRUE) {
    outport(CONTROL2+tbc_base, 0x0000);
  }
  else {
    outport(CONTROL2+tbc_base, 0x0200);
```

```
  }

  /* delay for reset to take effect */
  delay(50);

  /* write to Control Register 2 to deactivate reset */
  if (rs422_enable == TRUE) {
    outport(CONTROL2+tbc_base, 0x003C);
  }
  else {
    outport(CONTROL2+tbc_base, 0x023C);
  }

} /* end hard_reset() */


/
**********************************************************************
**

     Function: setbit()

      Summary: This procedure is used to set or clear a bit in an arbitrary
           length array.

        Usage: void setbit(array,bitnum,bit)
           unsigned short *array,bitnum,bit;

   Description: This procedure sets or clears a bit in an arbitrary length
           array. This procedure requires three parameters. The 'array'
           parameter indicates the array in which the bit is to be
           set or cleared. The 'bitnum' parameter indicates the bit
           number, from 0 .. n, of the bit which is to be set or
           cleared. The 'bit' parameter indicates whether the bit
           is to be set (1) or cleared (0).


**********************************************************************
***/

void setbit(array,bitnum,bit)
unsigned short *array,bitnum,bit;
{
```

```
   unsigned short mask,index;

   /* This routine sets a bit in an arbitrary length bit stream */

   /* see which 16-bit word the bit is in */
   index = bitnum / 16;

   /* create a bit mask into the 16-bit word */
   mask = 1 << (bitnum - (index * 16));

   if(bit)    /* set bit */
      array[index] |= mask;
   else       /* clear bit */
     array[index] &= ~mask;
}


/
**********************************************************************
**

     Function: getbit()

      Summary: This procedure is used to determine the value of a bit in
            an arbitrary length array.

        Usage: short getbit(array,bitnum)
            unsigned short *array,bitnum;


   Description: This procedure return the value of a specified bit in an
            arbitrary length array. This procedure requires two
            parameters. The 'array' parameter indicates the array in
            which the value of the bit is to be determined. The 'bitnum'
            parameter indicates the bit number, from 0 .. n, of the bit
            in which value is to be determined.


**********************************************************************
***/

short getbit(array,bitnum)
unsigned short *array,bitnum;
```

```
{
  unsigned short mask,index;

  /* This routine returns the value of a bit in an arbitrary length bit
    stream */

  /* see which 16-bit word the bit is in */
  index = bitnum / 16;

  /* create a bit mask into the 16-bit word */
  mask = 1 << (bitnum - (index * 16));

  if(array[index] & mask)
    return 1;
  else
    return 0;
}


/
**********************************************************************
**

    Function: enable_jtag_selftest()

     Summary: This procedure configures the PC-1149.1/100F for self-test
          operation using one of the four JTAG busses.

       Usage: void enable_jtag_selftest(test_bus)
          unsigned short test_bus;

  Description: This procedure configures the PC-1149.1/100F for self-test
          operation using one of the four JTAG busses. This procedure
          takes one parameter. The 'test_bus' parameter indicates
          which of the four JTAG busses will be enabled for self-test.

**********************************************************************
***/

void enable_jtag_selftest(test_bus)
unsigned short test_bus;
{
  unsigned short control2_data;
```

```
  /* get current Control Register 2 value by reading Status Register 2 */
  control2_data = inport(STATUS2+tbc_base);

  /* clear the self-test select bits in the data to be written to Control
    Register 2. */
  control2_data = control2_data & 0xE7FF;

  /* set the self-test select bits in the data to be written to Control
    Register 2 */
  if (test_bus <= JTAG_BUS4) {
    control2_data = control2_data | (test_bus << 11);
  }
  else {
    cputs("JTAG bus passed to procedure enable_jtag_selftest was invalid. \r\n");
  }

  /* set the self-test enable bit in the data to be written to Control
    Register 2 */
  control2_data = control2_data | 0x0400;

  /* write the data to Control Register 2 */
  outport(CONTROL2+tbc_base, control2_data);

} /* end enable_jtag_selftest() */


/
**********************************************************************
**

     Function: disable_jtag_selftest()

      Summary: This procedure disables the on-board self-test.

        Usage: void disable_jtag_selftest()

  Description: This procedure disables the on-board self-test by clearing
               the self-test enable bit in Control Register 2. No parameters
               are required.
```

```
**********************************************************************
***/

void disable_jtag_selftest()
{
  unsigned short control2_data;

  /* get current Control Register 2 value by reading Status Register 2 */
  control2_data = inport(STATUS2+tbc_base);

  /* clear the self-test select bits and the self-test enable bit in the
    data to be written to Control Register 2. */
  control2_data = control2_data & 0xE3FF;

  /* write the data to Control Register 2 */
  outport(CONTROL2+tbc_base, control2_data);

} /* end disable_jtag_selftest() */


/
**********************************************************************
**

     Function: scan_ir_turbo()

      Summary: Scans a bit stream into the TAP instruction register
            using turbo mode of operation.

        Usage: void scan_ir_turbo(test_bus, output, length, input)
            unsigned short test_bus;
            unsigned short *output;
            unsigned short length;
            unsigned short *input;

   Description: This routine scans an arbitrary length bit stream into a
            target TAP instruction register using the turbo mode of
            operation. This function has four parameters. The 'test_bus'
            parameter indicates which of the four JTAG busses will be
            used for the scan. The 'output' variable is an address to
            an array of byte to be scanned out the JTAG bus. The 'length'
            parameter specifies the length of the bitstream. The 'input'
            variable is an address to an array of byte values where the
```

data scanned in is to be stored.

Note: RX FIFO empty flag is asserted when 1 or more bytes is loaded into the RX FIFO. The RX FIFO cannot be read until two or more bytes have been received (RX is only read in word increments), therefore the RX FIFO empty flag cannot be used as an "RX data ready" indication.

```
*************************************************************************
***/

void scan_ir_turbo(test_bus, output, length, input)
unsigned short test_bus;
unsigned short *output;
unsigned short length;
unsigned short *input;
{
  unsigned char  cnt_status;        /* storage for status of 32-bit
                                       counter */
  unsigned short x;                 /* loop counter */
  unsigned char  shift_count;       /* holds right shift count in the
                                       event of odd bits */
  unsigned short psc_offset;        /* holds offset of PSC, based on JTAG
                                       bus being used */
  unsigned short tms_reg_offset;    /* offset of PSC TMS buffer/shifter,
                                       based on JTAG bus being used */
  unsigned short numwords = 0;      /* number of whole words to send,
                                       rounded up */
  unsigned short numsent = 0, numrcvd = 0;
                                    /* number of words sent, number of
                                       words received */

  /* setup for appropriate JTAG bus */
  if (test_bus == JTAG_BUS1) {
    psc_offset = PSC1_OFFSET;
    tms_reg_offset = TMS0_BUFFER;
  }
  else if (test_bus == JTAG_BUS2) {
    psc_offset = PSC1_OFFSET;
    tms_reg_offset = TMS1_BUFFER;
  }
  else if (test_bus == JTAG_BUS3) {
    psc_offset = PSC2_OFFSET;
```

```
  tms_reg_offset = TMS0_BUFFER;
}
else if (test_bus == JTAG_BUS4) {
  psc_offset = PSC2_OFFSET;
  tms_reg_offset = TMS1_BUFFER;
}

/* compute number of words to scan. get number of whole words and add
  1 to it if it does not fall on even word boundary */
numwords = length / 16;
if ((length % 16) != 0)
  ++numwords;

/* if length does not align to byte boundary, we have to shift right
  to move odd bits received into lsb's */
shift_count = (8 - (length % 8)) % 8;

/* verify turbo mode is off before starting */
outport(CONTROL2+tbc_base, (inport(STATUS2+tbc_base) & 0x1FFE));

/* remove TCK dependency on TDI and TDO buffers. restore dependency on
  TMS buffer */
if (tms_reg_offset == TMS0_BUFFER) {
  write_psc(psc_offset, MODE0, 0x30);
}
else {
  write_psc(psc_offset, MODE0, 0x28);
}

/* move TAP controller into SHIFT-IR state */
load_cnt_32(psc_offset, 5);
write_tms(psc_offset, tms_reg_offset, 0x06);

/* wait for 32-bit counter to reach terminal count */
do  {
  read_psc(psc_offset, MODE2, &cnt_status);
  cnt_status = cnt_status & 0x20;
}  while (cnt_status == 0);

/* restore TCK dependency on TDI and TDO buffers. remove dependency on
  TMS. set TMS high mode */
write_psc(psc_offset, MODE0, 0xE2);
```

```
/* load 32-bit counter with number of bits to scan */
load_cnt_32(psc_offset, (unsigned long) length);

/* enable turbo for the selected JTAG bus */
if (test_bus == JTAG_BUS1 || test_bus == JTAG_BUS2) {
  outport(CONTROL2+tbc_base, (inport(STATUS2+tbc_base) | 0x0003));
}
else {
  outport(CONTROL2+tbc_base, ((inport(STATUS2+tbc_base) & 0x1FFD) | 0x0001));
}

/* load transmit FIFO with as much data as possible up to quarter
  of FIFO size. */
x = 0;
while ((x < (TX_FIFO_SIZE/4)) && ((numwords - numsent) > 1)) {

  /* load word into transmit FIFO */
  outport(JTAG_TDO_FIFO+tbc_base, output[numsent]);

  /* increment number of words sent */
  numsent++;

  /* increment counter for next pass */
  x++;

}

/* scan all remaining words except last word */
while ((numwords - numsent) > 1) {

  /* wait for transmit FIFO to empty */
  while ((inport(STATUS3+tbc_base) & 0x0001) != 0x0000) {
  }

  /* wait for turbo to finish (*XDONE low) */
  while ((inport(STATUS2+tbc_base) & 0x4000) != 0x0000) {
  }

  /* load more data into transmit, upto a quarter of transmit size */
  x = 0;
  while ((x < (TX_FIFO_SIZE/4)) && ((numwords - numsent) > 1)) {

    /* load word into transmit FIFO */
```

```
    outport(JTAG_TDO_FIFO+tbc_base, output[numsent]);

    /* increment number of words sent */
    numsent++;

    /* increment counter for next pass */
    x++;

  }

  /* read data from receive FIFO, upto quarter of transmit size */
  x = 0;
  while ((x < TX_FIFO_SIZE/4) && ((numwords - numrcvd) > 1)) {

    /* get received word */
    input[numrcvd] = inport(JTAG_TDI_FIFO+tbc_base);

    /* increment received count */
    numrcvd++;

    /* increment counter for next pass */
    x++;

  }

}

/* load FIFO with the last word (or byte) */
if ((((length - 1) / 8) % 2) == 0) {
  outportb(JTAG_TDO_FIFO+tbc_base, get_byte_word(output[numsent], LSB));
}
else {
  outport(JTAG_TDO_FIFO+tbc_base, output[numsent]);
}

/* wait for transmit FIFO to empty */
while ((inport(STATUS3+tbc_base) & 0x0001) != 0x0000) {
}

/* wait for turbo to finish */
while((inport(STATUS2+tbc_base) & 0x4000) != 0x0000) {
}
```

```
  /* disable turbo operation */
  outport(CONTROL2+tbc_base, (inport(STATUS2+tbc_base) & 0x1FFE));

  /* read all remaining data from receive FIFO */
  while ((inport(STATUS3+tbc_base) & 0x0008) != 0x0000) {
    input[numrcvd] = inport(JTAG_TDI_FIFO+tbc_base);
    numrcvd++;
  }

  /* shift bits in last word as necessary */
  if ((((length - 1) / 8) % 2) == 0) {
    input[numwords-1] = (input[numwords-1] & 0x00FF) >> shift_count;
  }
  else {
    input[numwords-1] = (input[numwords-1] & 0x00FF) |
                ((input[numwords-1] >> shift_count) & 0xFF00);
  }

  /* restore TCK dependency on TMS buffer. remove TCK dependency on TDI
    and TDO buffers. */
  if (tms_reg_offset == TMS0_BUFFER) {
    write_psc(psc_offset, MODE0, 0x30);
  }
  else {
    write_psc(psc_offset, MODE0, 0x28);
  }

  /* load TMS buffer to move TAP controller to RUN-TEST/IDLE state */
  load_cnt_32(psc_offset, 2);
  write_tms(psc_offset, tms_reg_offset, 0x01);

  /* wait for 32-bit counter to reach terminal count */
  do  {
    read_psc(psc_offset, MODE2, &cnt_status);
    cnt_status = cnt_status & 0x20;
  }  while (cnt_status == 0);

} /* end scan_ir_turbo() */


/
**********************************************************************
**
```

Function: scan_dr_turbo()

Summary: Scans a bit stream into a TAP data register using turbo mode of operation.

Usage: void scan_dr_turbo(test_bus, output, length, input)
unsigned short test_bus;
unsigned short *output;
unsigned short length;
unsigned short *input;

Description: This routine scans an arbitrary length bit stream into a target TAP data register using the turbo mode of operation. This function has four parameters. The 'test_bus' parameter indicates which of the four JTAG busses will be used for the scan. The 'output' variable is an address to an array of byte to be scanned out the JTAG bus. The 'length' parameter specifies the length of the bitstream. The 'input' variable is an address to an array of byte values where the data scanned in is to be stored.

Note: RX FIFO empty flag is asserted when 1 or more bytes is loaded into the RX FIFO. The RX FIFO cannot be read until two or more bytes have been received (RX is only read in word increments), therefore the RX FIFO empty flag cannot be used as an "RX data ready" indication.

```
**********************************************************************
***/

void scan_dr_turbo(test_bus, output, length, input)
unsigned short test_bus;
unsigned short *output;
unsigned short length;
unsigned short *input;
{
  unsigned char  cnt_status;        /* storage for status of 32-bit
                                counter */
  unsigned short x;                 /* loop counter */
  unsigned char  shift_count;       /* holds right shift count in the
                                event of odd bits */
  unsigned short psc_offset;        /* holds offset of PSC, based on JTAG
```

```
                        bus being used */
unsigned short tms_reg_offset;    /* offset of PSC TMS buffer/shifter,
                        based on JTAG bus being used */
unsigned short numwords = 0;      /* number of whole words to send,
                        rounded up */
unsigned short numsent = 0, numrcvd = 0;
                        /* number of words sent, number of
                        words received */

/* setup for appropriate JTAG bus */
if (test_bus == JTAG_BUS1) {
  psc_offset = PSC1_OFFSET;
  tms_reg_offset = TMS0_BUFFER;
}
else if (test_bus == JTAG_BUS2) {
  psc_offset = PSC1_OFFSET;
  tms_reg_offset = TMS1_BUFFER;
}
else if (test_bus == JTAG_BUS3) {
  psc_offset = PSC2_OFFSET;
  tms_reg_offset = TMS0_BUFFER;
}
else if (test_bus == JTAG_BUS4) {
  psc_offset = PSC2_OFFSET;
  tms_reg_offset = TMS1_BUFFER;
}

/* compute number of words to scan. get number of whole words and add
  1 to it if it does not fall on even word boundary */
numwords = length / 16;
if ((length % 16) != 0)
  ++numwords;

/* if length does not align to byte boundary, we have to shift right
  to move odd bits received into lsb's */
shift_count = (8 - (length % 8)) % 8;

/* verify turbo mode is off before starting */
outport(CONTROL2+tbc_base, (inport(STATUS2+tbc_base) & 0x1FFE));

/* remove TCK dependency on TDI and TDO buffers. restore dependency on
  TMS buffer */
if (tms_reg_offset == TMS0_BUFFER) {
```

```
  write_psc(psc_offset, MODE0, 0x30);
}
else {
  write_psc(psc_offset, MODE0, 0x28);
}

/* move TAP controller into SHIFT-DR state */
load_cnt_32(psc_offset, 4);
write_tms(psc_offset, tms_reg_offset, 0x02);

/* wait for 32-bit counter to reach terminal count */
do  {
  read_psc(psc_offset, MODE2, &cnt_status);
  cnt_status = cnt_status & 0x20;
}  while (cnt_status == 0);

/* restore TCK dependency on TDI and TDO buffers. remove dependency on
  TMS. set TMS high mode */
write_psc(psc_offset, MODE0, 0xE2);

/* load 32-bit counter with number of bits to scan */
load_cnt_32(psc_offset, (unsigned long) length);

/* enable turbo for the selected JTAG bus */
if (test_bus == JTAG_BUS1 || test_bus == JTAG_BUS2) {
  outport(CONTROL2+tbc_base, (inport(STATUS2+tbc_base) | 0x0003));
}
else {
  outport(CONTROL2+tbc_base, ((inport(STATUS2+tbc_base) & 0x1FFD) | 0x0001));
}

/* load transmit FIFO with as much data as possible up to quarter
  of FIFO size. */
x = 0;
while ((x < (TX_FIFO_SIZE/4)) && ((numwords - numsent) > 1)) {

  /* load word into transmit FIFO */
  outport(JTAG_TDO_FIFO+tbc_base, output[numsent]);

  /* increment number of words sent */
  numsent++;

  /* increment counter for next pass */
```

```
    x++;

  }

  /* scan all remaining words except last word */
  while ((numwords - numsent) > 1) {

    /* wait for transmit FIFO to empty */
    while ((inport(STATUS3+tbc_base) & 0x0001) != 0x0000) {
    }

    /* wait for turbo to finish (*XDONE low) */
    while ((inport(STATUS2+tbc_base) & 0x4000) != 0x0000) {
    }

    /* load more data into transmit, upto a quarter of transmit size */
    x = 0;
    while ((x < (TX_FIFO_SIZE/4)) && ((numwords - numsent) > 1)) {

      /* load word into transmit FIFO */
      outport(JTAG_TDO_FIFO+tbc_base, output[numsent]);

      /* increment number of words sent */
      numsent++;

      /* increment counter for next pass */
      x++;

    }

    /* read data from receive FIFO, upto quarter of transmit size */
    x = 0;
    while ((x < TX_FIFO_SIZE/4) && ((numwords - numrcvd) > 1)) {

      /* get received word */
      input[numrcvd] = inport(JTAG_TDI_FIFO+tbc_base);

      /* increment received count */
      numrcvd++;

      /* increment counter for next pass */
      x++;
```

```
  }

}

/* load FIFO with the last word (or byte) */
if ((((length - 1) / 8) % 2) == 0) {
  outportb(JTAG_TDO_FIFO+tbc_base, get_byte_word(output[numsent], LSB));
}
else {
  outport(JTAG_TDO_FIFO+tbc_base, output[numsent]);
}

/* wait for transmit FIFO to empty */
while ((inport(STATUS3+tbc_base) & 0x0001) != 0x0000) {
}

/* wait for turbo to finish */
while((inport(STATUS2+tbc_base) & 0x4000) != 0x0000) {
}

/* disable turbo operation */
outport(CONTROL2+tbc_base, (inport(STATUS2+tbc_base) & 0x1FFE));

/* read all remaining data from receive FIFO */
while ((inport(STATUS3+tbc_base) & 0x0008) != 0x0000) {
  input[numrcvd] = inport(JTAG_TDI_FIFO+tbc_base);
  numrcvd++;
}

/* shift bits in last word as necessary */
if ((((length - 1) / 8) % 2) == 0) {
  input[numwords-1] = (input[numwords-1] & 0x00FF) >> shift_count;
}
else {
  input[numwords-1] = (input[numwords-1] & 0x00FF) |
                ((input[numwords-1] >> shift_count) & 0xFF00);
}

/* restore TCK dependency on TMS buffer. remove TCK dependency on TDI
  and TDO buffers. */
if (tms_reg_offset == TMS0_BUFFER) {
  write_psc(psc_offset, MODE0, 0x30);
}
```

```
  else {
    write_psc(psc_offset, MODE0, 0x28);
  }

  /* load TMS buffer to move TAP controller to RUN-TEST/IDLE state */
  load_cnt_32(psc_offset, 2);
  write_tms(psc_offset, tms_reg_offset, 0x01);

  /* wait for 32-bit counter to reach terminal count */
  do  {
    read_psc(psc_offset, MODE2, &cnt_status);
    cnt_status = cnt_status & 0x20;
  }  while (cnt_status == 0);

} /* end scan_dr_turbo() */


/
**********************************************************************
**

     Function: jtag_selftest()

      Summary: Tests all of the JTAG busses using the self-test mode of
               operation on the PC-1149.1/100F.

       Usage: unsigned short jtag_selftest()

   Description: This function self-tests the JTAG busses on the
                PC-1149.1/100F using the self-test mode of operation.
                No parameters are required.


**********************************************************************
***/

unsigned short jtag_selftest()
{
   unsigned short result = PASS;      /* holds pass/fail status of tst */
   unsigned short i;                  /* loop counter */
   unsigned short jtag_data_out[3], jtag_data_in[3];
                                 /* storage for scan output and input */
   unsigned short parallel_rdbk;      /* holds data from parallel rdbk */
```

```
unsigned short jtag_bus[4]  = { JTAG_BUS1,
                        JTAG_BUS2,
                        JTAG_BUS3,
                        JTAG_BUS4 };  /* table of jtag busses to
                                test */
unsigned short psc_offset[4] = { PSC1_OFFSET,
                        PSC1_OFFSET,
                        PSC2_OFFSET,
                        PSC2_OFFSET };/* table of offsets of
                                parallel/serial
                                converter of bus
                                under test */
unsigned short mode0_data[4] = { 0x30,
                        0x28,
                        0x30,
                        0x28 };     /* table of data written
                                to Mode 0 register of
                                of parallel/serial
                                converter during
                                TEST-LOGIC/RESET */

unsigned short tms_reg_offset[4] = { TMS0_BUFFER,
                          TMS1_BUFFER,
                          TMS0_BUFFER,
                          TMS1_BUFFER };
                                /* offset of TMS buffer
                                  for bus under test */

cputs("   Performing JTAG self-test (non-turbo) ..... ");


/* ensure parallel outputs are not tri-stated */
outport(CONTROL2+tbc_base, (inport(STATUS2+tbc_base) & 0xFEFF));

/* set all parallel outputs to a low state */
outport(PARALLEL_OUT+tbc_base, 0);

for (i = 0; i < 4; i++)  {

  /* enable self-test of JTAG bus */
  enable_jtag_selftest(jtag_bus[i]);

  /* do a TEST-LOGIC/RESET */
```

```
write_psc(psc_offset[i], MODE0, mode0_data[i]);
load_cnt_32(psc_offset[i], 6);
write_tms(psc_offset[i], tms_reg_offset[i], 0xFF);

/* do a default BYPASS scan after TEST-LOGIC/RESET */
jtag_data_in[0] = 0;
jtag_data_in[1] = 0;
jtag_data_in[2] = 0;
jtag_data_out[0] = 0x1234;
jtag_data_out[1] = 0x5678;
jtag_data_out[2] = 0x9ABC;
scan_dr(jtag_bus[i], jtag_data_out, 48, jtag_data_in);

/* if data scanned is not correct, print failure messages and set test
   test status to failed */
if ((jtag_data_in[0] != 0x2468) || (jtag_data_in[1] != 0xACF0) ||
   (jtag_data_in[2] != 0x3578)) {

   /* print failure messages */
   cprintf("\r\n\r\n");
   cprintf("Failure on scan of BYPASS register after TEST-LOGIC/RESET \r\n");
   cprintf("on JTAG Bus: %04X.\r\n", i+1);
   cprintf("Expected DR response word 1 = %04X\r\n", 0x2468);
   cprintf("Actual DR response word 1   = %04X\r\n", jtag_data_in[0]);
   cprintf("Expected DR response word 2 = %04X\r\n", 0xACF0);
   cprintf("Actual DR response word 2   = %04X\r\n", jtag_data_in[1]);
   cprintf("Expected DR response word 3 = %04X\r\n", 0x3578);
   cprintf("Actual DR response word 3   = %04X\r\n\r\n", jtag_data_in[2]);

   /* indicate test failed */
   result = FAIL;

   /* exit test */
   break;
}

/* load BYPASS instruction */
jtag_data_in[0] = 0;
jtag_data_out[0] = BYPASS_18374;
scan_ir(jtag_bus[i], jtag_data_out, ON_BOARD_IR_LEN, jtag_data_in);

/* if instruction capture not correct, print failure messages and set test
   test status to failed */
```

```
if ((jtag_data_in[0] & 0x00FF) != 0x001D) {

  /* print failure messages */
  cprintf("\r\n\r\n");
  cprintf("Failure on instruction capture while loading BYPASS \r\n");
  cprintf("instruction on JTAG Bus: %04X.\r\n", i+1);
  cprintf("Expected IR response byte = %02X\r\n", 0x1D);
  cprintf("Actual IR response byte   = %02X\r\n", jtag_data_in[0]);

  /* indicate test failed */
  result = FAIL;

  /* exit test */
  break;

}

/* scan data through BYPASS register */
jtag_data_in[0] = 0;
jtag_data_in[1] = 0;
jtag_data_in[2] = 0;
jtag_data_out[0] = 0x1234;
jtag_data_out[1] = 0x5678;
jtag_data_out[2] = 0x9ABC;
scan_dr(jtag_bus[i], jtag_data_out, 48, jtag_data_in);

/* if data scanned is not correct, print failure messages and set test
  test status to failed */
if ((jtag_data_in[0] != 0x2468) || (jtag_data_in[1] != 0xACF0) ||
  (jtag_data_in[2] != 0x3578)) {

  /* print failure messages */
  cprintf("\r\n\r\n");
  cprintf("Failure on scan of BYPASS register after loading \r\n");
  cprintf("BYPASS instruction on JTAG Bus: %04X.\r\n", i+1);
  cprintf("Expected DR response word 1 = %04X\r\n", 0x2468);
  cprintf("Actual DR response word 1 = %04X\r\n", jtag_data_in[0]);
  cprintf("Expected DR response word 2 = %04X\r\n", 0xACF0);
  cprintf("Actual DR response word 2 = %04X\r\n", jtag_data_in[1]);
  cprintf("Expected DR response word 3 = %04X\r\n", 0x3578);
  cprintf("Actual DR response word 3 = %04X\r\n\r\n", jtag_data_in[2]);

  /* indicate test failed */
```

```
  result = FAIL;

  /* exit test */
  break;

}

/* load EXTEST instruction */
jtag_data_in[0] = 0;
jtag_data_out[0] = EXTEST_18374;
scan_ir(jtag_bus[i], jtag_data_out, ON_BOARD_IR_LEN, jtag_data_in);

/* if instruction capture not correct, print failure messages and set test
  test status to failed */
if ((jtag_data_in[0] & 0x00FF) != 0x001D) {

  /* print failure messages */
  cprintf("\r\n\r\n");
  cprintf("Failure on instruction capture while loading EXTEST \r\n");
  cprintf("instruction on JTAG Bus: %04X.\r\n", i+1);
  cprintf("Expected IR response byte = %02X\r\n", 0x1D);
  cprintf("Actual IR response byte   = %02X\r\n", jtag_data_in[0]);

  /* indicate test failed */
  result = FAIL;

  /* exit test */
  break;

}

/* scan data in to drive parallel outputs to $5555 */
jtag_data_in[0] = 0;
jtag_data_in[1] = 0;
jtag_data_in[2] = 0;
jtag_data_out[0] = 0xA954;
jtag_data_out[1] = 0x0002;
jtag_data_out[2] = 0x0090;
scan_dr(jtag_bus[i], jtag_data_out, ON_BOARD_DR_LEN, jtag_data_in);

/* read parallel readback */
parallel_rdbk = inport(PARALLEL_OUT_RDBK+tbc_base);
```

```
/* if data read from parallel readback is not same as expected, print
   failure message and set test status to failed */
if (parallel_rdbk != 0x5555) {

   /* print failure messages */
   cprintf("\r\n\r\n");
   cprintf("Failure on parallel readback after loading EXTEST \r\n");
   cprintf("instruction on JTAG Bus: %04X.\r\n", i+1);
   cprintf("Expected Resp     = %04X\r\n", 0x5555);
   cprintf("Actual Input Resp = %04X\r\n", parallel_rdbk);

   /* indicate test failed */
   result = FAIL;

   /* exit test */
   break;

}

/* scan data in to drive parallel outputs to $AAAA */
jtag_data_in[0] = 0;
jtag_data_in[1] = 0;
jtag_data_in[2] = 0;
jtag_data_out[0] = 0x54AA;
jtag_data_out[1] = 0x0001;
jtag_data_out[2] = 0x0090;
scan_dr(jtag_bus[i], jtag_data_out, ON_BOARD_DR_LEN, jtag_data_in);

/* read parallel readback */
parallel_rdbk = inport(PARALLEL_OUT_RDBK+tbc_base);

/* if data read from parallel readback is not same as expected, print
   failure message and set test status to failed */
if (parallel_rdbk != 0xAAAA) {

   /* print failure messages */
   cprintf("\r\n\r\n");
   cprintf("Failure on parallel readback after loading EXTEST \r\n");
   cprintf("instruction on JTAG Bus: %04X.\r\n", i+1);
   cprintf("Expected Resp     = %04X\r\n", 0xAAAA);
   cprintf("Actual Input Resp = %04X\r\n", parallel_rdbk);

   /* indicate test failed */
```

```
        result = FAIL;

        /* exit test */
        break;

      }

      /* load SAMPLE/PRELOAD instruction into on-board SCAN18374 */
      jtag_data_out[0] = SAMPLE_18374;
      scan_ir(jtag_bus[i], jtag_data_out, ON_BOARD_IR_LEN, jtag_data_in);

    }

    /* disable the self-test function */
    disable_jtag_selftest();

    /* disable parallel outputs */
    outport(CONTROL2+tbc_base, (inport(STATUS2+tbc_base) | 0x0100));

    /* print out the test results */
    if (result == PASS) {
      textcolor(LIGHTGREEN);
      cputs("test passed. \r\n");
      textcolor(WHITE);
    }

    /* return test results */
    return result;

} /* end jtag_selftest() */


/
**********************************************************************
**

     Function: jtag_selftest_turbo()

      Summary: Tests all of the JTAG busses using the self-test mode of
               operation on the PC-1149.1/100F.

        Usage: unsigned short jtag_selftest()
```

```
   Description: This function tests the JTAG busses on the PC-1149.1/100F
                using the self-test mode of operation. This procedure
                uses turbo mode for all scan operations. No parameters are
                required.


**********************************************************************
***/

unsigned short jtag_selftest_turbo()
{
   unsigned short result = PASS;      /* holds pass/fail status of tst */
   unsigned short i;                  /* loop counter */
   unsigned short jtag_data_out[3], jtag_data_in[3];
                              /* storage for scan output and input */
   unsigned short parallel_rdbk;      /* holds data from parallel rdbk */
   unsigned short jtag_bus[4]   = { JTAG_BUS1,
                           JTAG_BUS2,
                           JTAG_BUS3,
                           JTAG_BUS4 };  /* table of jtag busses to
                                    test */
   unsigned short psc_offset[4] = { PSC1_OFFSET,
                           PSC1_OFFSET,
                           PSC2_OFFSET,
                           PSC2_OFFSET };/* table of offsets of
                                    parallel/serial
                                    converter of bus
                                    under test */
   unsigned short mode0_data[4] = { 0x30,
                           0x28,
                           0x30,
                           0x28 };      /* table of data written
                                    to Mode 0 register of
                                    of parallel/serial
                                    converter during
                                    TEST-LOGIC/RESET */

   unsigned short tms_reg_offset[4] = { TMS0_BUFFER,
                              TMS1_BUFFER,
                              TMS0_BUFFER,
                              TMS1_BUFFER };
                                    /* offset of TMS buffer
                                      for bus under test */
```

```
cputs("   Performing JTAG self-test (turbo) ..... ");


/* ensure parallel outputs are not tri-stated */
outport(CONTROL2+tbc_base, (inport(STATUS2+tbc_base) & 0xFEFF));

/* set all parallel outputs to a low state */
outport(PARALLEL_OUT+tbc_base, 0);

for (i = 0; i < 4; i++)  {

  /* enable self-test of JTAG bus */
  enable_jtag_selftest(jtag_bus[i]);

  /* do a TEST-LOGIC/RESET */
  write_psc(psc_offset[i], MODE0, mode0_data[i]);
  load_cnt_32(psc_offset[i], 6);
  write_tms(psc_offset[i], tms_reg_offset[i], 0xFF);

  /* do a default BYPASS scan after TEST-LOGIC/RESET */
  jtag_data_in[0] = 0;
  jtag_data_in[1] = 0;
  jtag_data_in[2] = 0;
  jtag_data_out[0] = 0x1234;
  jtag_data_out[1] = 0x5678;
  jtag_data_out[2] = 0x9ABC;
  scan_dr_turbo(jtag_bus[i], jtag_data_out, 48, jtag_data_in);

  /* if data scanned is not correct, print failure messages and set test
    test status to failed */
  if ((jtag_data_in[0] != 0x2468) || (jtag_data_in[1] != 0xACF0) ||
    (jtag_data_in[2] != 0x3578)) {

    /* print failure messages */
    cprintf("\r\n\r\n");
    cprintf("Failure on scan of BYPASS register after TEST-LOGIC/RESET \r\n");
    cprintf("on JTAG Bus: %04X.\r\n", i+1);
    cprintf("Expected DR response word 1 = %04X\r\n", 0x2468);
    cprintf("Actual DR response word 1   = %04X\r\n", jtag_data_in[0]);
    cprintf("Expected DR response word 2 = %04X\r\n", 0xACF0);
    cprintf("Actual DR response word 2   = %04X\r\n", jtag_data_in[1]);
    cprintf("Expected DR response word 3 = %04X\r\n", 0x3578);
```

```
  cprintf("Actual DR response word 3   = %04X\r\n\r\n", jtag_data_in[2]);

  /* indicate test failed */
  result = FAIL;

  /* exit test */
  break;
}

/* load BYPASS instruction */
jtag_data_in[0] = 0;
jtag_data_out[0] = BYPASS_18374;
scan_ir_turbo(jtag_bus[i], jtag_data_out, ON_BOARD_IR_LEN, jtag_data_in);

/* if instruction capture not correct, print failure messages and set test
  test status to failed */
if ((jtag_data_in[0] & 0x00FF) != 0x001D) {

  /* print failure messages */
  cprintf("\r\n\r\n");
  cprintf("Failure on instruction capture while loading BYPASS \r\n");
  cprintf("instruction on JTAG Bus: %04X.\r\n", i+1);
  cprintf("Expected IR response byte = %02X\r\n", 0x1D);
  cprintf("Actual IR response byte   = %02X\r\n", jtag_data_in[0]);

  /* indicate test failed */
  result = FAIL;

  /* exit test */
  break;

}

/* scan data through BYPASS register */
jtag_data_in[0] = 0;
jtag_data_in[1] = 0;
jtag_data_in[2] = 0;
jtag_data_out[0] = 0x1234;
jtag_data_out[1] = 0x5678;
jtag_data_out[2] = 0x9ABC;
scan_dr_turbo(jtag_bus[i], jtag_data_out, 48, jtag_data_in);

/* if data scanned is not correct, print failure messages and set test
```

```
  test status to failed */
if ((jtag_data_in[0] != 0x2468) || (jtag_data_in[1] != 0xACF0) ||
  (jtag_data_in[2] != 0x3578)) {

  /* print failure messages */
  cprintf("\r\n\r\n");
  cprintf("Failure on scan of BYPASS register after loading \r\n");
  cprintf("BYPASS instruction on JTAG Bus: %04X.\r\n", i+1);
  cprintf("Expected DR response word 1 = %04X\r\n", 0x2468);
  cprintf("Actual DR response word 1 = %04X\r\n", jtag_data_in[0]);
  cprintf("Expected DR response word 2 = %04X\r\n", 0xACF0);
  cprintf("Actual DR response word 2 = %04X\r\n", jtag_data_in[1]);
  cprintf("Expected DR response word 3 = %04X\r\n", 0x3578);
  cprintf("Actual DR response word 3 = %04X\r\n\r\n", jtag_data_in[2]);

  /* indicate test failed */
  result = FAIL;

  /* exit test */
  break;

}

/* load EXTEST instruction */
jtag_data_in[0] = 0;
jtag_data_out[0] = EXTEST_18374;
scan_ir_turbo(jtag_bus[i], jtag_data_out, ON_BOARD_IR_LEN, jtag_data_in);

/* if instruction capture not correct, print failure messages and set test
  test status to failed */
if ((jtag_data_in[0] & 0x00FF) != 0x001D) {

  /* print failure messages */
  cprintf("\r\n\r\n");
  cprintf("Failure on instruction capture while loading EXTEST \r\n");
  cprintf("instruction on JTAG Bus: %04X.\r\n", i+1);
  cprintf("Expected IR response byte = %02X\r\n", 0x1D);
  cprintf("Actual IR response byte   = %02X\r\n", jtag_data_in[0]);

  /* indicate test failed */
  result = FAIL;

  /* exit test */
```

```
  break;

}

/* scan data in to drive parallel outputs to $5555 */
jtag_data_in[0] = 0;
jtag_data_in[1] = 0;
jtag_data_in[2] = 0;
jtag_data_out[0] = 0xA954;
jtag_data_out[1] = 0x0002;
jtag_data_out[2] = 0x0090;
scan_dr_turbo(jtag_bus[i], jtag_data_out, ON_BOARD_DR_LEN, jtag_data_in);

/* read parallel readback */
parallel_rdbk = inport(PARALLEL_OUT_RDBK+tbc_base);

/* if data read from parallel readback is not same as expected, print
  failure message and set test status to failed */
if (parallel_rdbk != 0x5555) {

  /* print failure messages */
  cprintf("\r\n\r\n");
  cprintf("Failure on parallel readback after loading EXTEST \r\n");
  cprintf("instruction on JTAG Bus: %04X.\r\n", i+1);
  cprintf("Expected Resp     = %04X\r\n", 0x5555);
  cprintf("Actual Input Resp = %04X\r\n", parallel_rdbk);

  /* indicate test failed */
  result = FAIL;

  /* exit test */
  break;

}

/* scan data in to drive parallel outputs to $AAAA */
jtag_data_in[0] = 0;
jtag_data_in[1] = 0;
jtag_data_in[2] = 0;
jtag_data_out[0] = 0x54AA;
jtag_data_out[1] = 0x0001;
jtag_data_out[2] = 0x0090;
scan_dr_turbo(jtag_bus[i], jtag_data_out, ON_BOARD_DR_LEN, jtag_data_in);
```

```
      /* read parallel readback */
      parallel_rdbk = inport(PARALLEL_OUT_RDBK+tbc_base);

      /* if data read from parallel readback is not same as expected, print
        failure message and set test status to failed */
      if (parallel_rdbk != 0xAAAA) {

        /* print failure messages */
        cprintf("\r\n\r\n");
        cprintf("Failure on parallel readback after loading EXTEST \r\n");
        cprintf("instruction on JTAG Bus: %04X.\r\n", i+1);
        cprintf("Expected Resp     = %04X  \r\n", 0xAAAA);
        cprintf("Actual Input Resp = %04X  \r\n", parallel_rdbk);

        /* indicate test failed */
        result = FAIL;

        /* exit test */
        break;
      }

      /* load SAMPLE/PRELOAD instruction into on-board SCAN18374 */
      jtag_data_out[0] = SAMPLE_18374;
      scan_ir_turbo(jtag_bus[i], jtag_data_out, ON_BOARD_IR_LEN, jtag_data_in);
    }

    /* disable the self-test function */
    disable_jtag_selftest();

    /* disable parallel outputs */
    outport(CONTROL2+tbc_base, (inport(STATUS2+tbc_base) | 0x0100));

    /* print out the test results */
    if (result == PASS) {
      textcolor(LIGHTGREEN);
      cputs("test passed. \r\n");
      textcolor(WHITE);
    }

    /* return test results */
    return result;

} /* end jtag_selftest_turbo() */
```

D.19 sim.h

```
/* This file is to be used when testing with the  Verilog model */

/*currently writes two files: tms and tdi, to be read by
verilog into memories and fed into the simulated chip.
verilog will return bits in a new file, tdo_in which
contains the bits returned on the tdo.  issuing a start
command gets verilog running */


enum ADDR{
  TDI_ADDR = 0x120,
  /* for writing to the tdi stream*/
  TMS_ADDR,
  /*for writing to the tms stream*/
  COMMAND,
  /*for writing commands*/
  TDO_ADDR = 0x120,
  /*for reading bits returned by tdo*/
  STATUS = 0x122};
  /*for checking board status*/

enum B_STAT{
  DONE,
  NOT_DONE
  };

enum COM{
  START,
  /*start sending bits, closes files, tells verilog to go*/
  STOP,
  /*must be called after start, and before the next start*/
  RESET
    /*must be called on startup to clear FIFOs (well, actually
      to opne the fifo files*/
  };

enum JTAG_STATES{
  TLR,
  RTI,
  SDRS,
```

```
 CDR,
 SDR,
 E1DR,
 PDR,
 E2DR,
 UDR,
 SIRS,
 CIR,
 SIR,
 E1IR,
 PIR,
 E2IR,
 UIR
 };


void board_write(enum ADDR address, short byte);
/*writes a byte to a board register*/

/* can write data to tdi or tms, or commands,
   start, stop, and reset*/

int board_read(enum ADDR address, short *byte);
/*reads a byte from a board register*/
/*can read status, or tdo*/

void do_command(enum COM c);
/*opens and closes files */
```

D.20 sim.c

```
/* Testing file - for Verilog simulation */

#include "ancis.h"
#include <time.h>

extern  FILE *logf;
extern  int logf_level;

enum B_STAT board_status = DONE;

short bytes_returned = 0;
FILE *tms_file, *tdi_file, *tdo_file, *size_file, *done_file,
  *cycle_file;
int size_data_in_fifo;
int cycles = 0;

void board_write(enum ADDR address, short byte)
{
 int i;

 if(TEST>6)
   printf("            board_write addr:%x  data:%x \n", address, byte);

/*deal with what board_status should be*/

 if(address == TDI_ADDR)
   {
    board_status = NOT_DONE;
    fprintf(tdi_file, "%02x ", byte);
    size_data_in_fifo++;
   }

 if(address == TMS_ADDR)
   {
    board_status = NOT_DONE;
    fprintf(tms_file, "%02x ", byte);
    }

 if(address == COMMAND)
  do_command(byte);
```

```
}

int board_read(enum ADDR address, short *byte)
{
 int i;
 int c;

 /*returns size of data written to byte*/
 /*returns 0 if no bytes left*/
 /*once you start reading the tdo you must keep reading
  till you've got all of it*/

 if(address == TDO_ADDR)
  {
   /*rad bytes from tdo_file, they are in hex
       so you need 4 for a byte*/

   /*WHAT DIRECTION DO THEY COME BACK IN>> left
       in the same direciton as they are read out of the
       file*/
   /*clear byte*/

   *byte = 0;

   for(i=1; i>=0; i--)
       {
        c = getc(tdo_file);

        /*there are returns between each byte*/
        if (c == '\n')
         c = getc(tdo_file);

        /*end if c = EOF*/
        if (c == EOF)
         return( (i-1) * -4);

        /*convert c from hex characters*/
        c = c - '0';
        if (c > 9)
         c = c-39;

        /*add c to byte*/
```

```
        *byte = *byte | (c << (i*4));
       }
    if (TEST>6)
       printf("          board_read: %x\n", *byte);

    return(8);
   }

 if(address == STATUS)
   {
    *byte = board_status;
    if (TEST>6)
       printf("          board_read: status\n");

    return(0);
   }

}

void do_command(enum COM c)
{
 if (c == RESET)
   {
    /*open files for writing to*/
    /*for now take whatever bits are written, and write them to
       files to be sent to verilog as the TMS stream and the
       TDI stream */

    /*open files*/
    if ((tms_file = fopen("fifo_tms_out", "w")) == NULL)
       {
        printf("can't open file tms_out\n");
        if (logf_level>0)
          fprintf(logf, "can't open file tms_out\n");
        return;
       }

    if ((tdi_file = fopen("fifo_tdi_out", "w")) == NULL)
       {
        printf("can't open file tdi_out\n");
        if (logf_level>0)
          fprintf(logf, "can't open file tdi_out\n");
        return;
```

```
        }

    bytes_returned =0;
    board_status = DONE;
    size_data_in_fifo = 0;
   }

else if (c == START)
   {
    board_status = DONE;
    /*close files*/
    fclose(tdi_file);
    fclose(tms_file);
    fclose(tdo_file);

    /*open file to place file size in, for verilog to read*/

    /*reopen files*/
    if ((size_file = fopen("file_size", "w")) == NULL)
        {
         printf("can't open file file_size\n");
         if (logf_level>0)
           fprintf(logf, "can't open file file_size\n");
         return;
        }
    fprintf(size_file, "%02x\n", size_data_in_fifo);
    fclose(size_file);

    if (TEST>5)
        printf("----------------Verilog is running----------------\n");
    /*tell verilog to go*/
    cycle_file = fopen("cycle_file", "w");
    fprintf(cycle_file, "%04x\n", cycles);
    cycles++;
    fclose(cycle_file);

    /*wait till verilog is done*/
    /*verilog is done reading /writting when it writes the
        file "done_file", wait for it to be readable*/

    while((done_file = fopen("done_file", "r")) == NULL)
        {
         /*wait for a bit (10 seconds)*/
```

```
        sleep(10);
       }

   /* delete done_file so it's not around 'till verilog
        writes it again*/

   remove("done_file");
   fclose(done_file);

   size_data_in_fifo = 0;
  }

 else if (c == STOP)
  {
   board_status = DONE;

   /*reopen files*/
   if ((tms_file = fopen("fifo_tms_out", "w")) == NULL)
       {
        printf("can't open file tms_out\n");
        if (logf_level>0)
          fprintf(logf, "can't open file tms_out\n");
        return;
       }

   if ((tdi_file = fopen("fifo_tdi_out", "w")) == NULL)
       {
        printf("can't open file tdi_out\n");
        if (logf_level>0)
          fprintf(logf, "can't open file tdi_out\n");
        return;
       }

   if ((tdo_file = fopen("fifo_tdo_in", "r")) == NULL)
       {
        printf("meep, can't open file tdo_in \n");
        if (logf_level>0)
          fprintf(logf, "meep, can't open file tdo_in \n");
        while(1 == 1)
          sleep(100);
       }
  }
}
```