TEXAS INSTRUMENTS

TSB43AA82

iSphynxII Lynxsoft SBP-2 Target Firmware Programmer's Guide

User's Guide

June 2003

Catalog Interface Solutions

SLLU061

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Table Of Contents

Table of Figures

1. Related Documentation

This section provides a list, including title, date, and publisher of documents that might be referenced or needed during code development.

- Serial Bus Protocol 2 (SBP-2). ANSI NCITS 325-1998.
- Information Technology – Reduced Block Commands (RBC) ANSI NCITS 330-1999.
- SCSI-3 Primary Commands ANSI NCITS X3.301-199x
- IEEE Standard for a High Performance Serial Bus. IEEE Std 1394-1995.
- IEEE Standard for a High Performance Serial Bus. IEEE Std 1394a-2000.
- TSB43AA82 Data Manual (SLLS512)
- TSB43AA82A Data Manual (SLLS512A)

2. Introduction

This Programmer's Guide provides the software developer information on the iSphynxII (TSB43AA82) Lynxsoft firmware architecture, programming interfaces, and how to utilize the code to develop an SBP-2 application. The document describes the basic architecture of the TSB43AA82 firmware, some sample applications, the Target Applications Programming Interface (TAPI), and the Hardware Abstraction Layer (HAL).

This document assumes that the reader is or will become familiar with IEEE standards 1394-1995, 1394a-2000 and ANSI NCITS 325-1998 (SBP-2). Readers may also find that having a printed copy of the data manual for the TSB43AA82 a convenient reference while considering the details of the function prototypes and data structures presented here.

Two application examples are provided that utilize the lower layer firmware and provide example code. One application ("Monitor") is a basic node on the 1394 bus with a user interface. This allows the developer to become familiar with the capabilities of the TSB43AA82 and operations on the bus, as well as understand the operation of a node connected to the bus. The other application ("HDD") provides the basics of a hard disk drive using reduced block commands (RBC). This application will emulate a HDD using on-board memory or can be made to use a real HDD if the hardware is in place to utilize it.

The **Target Application Programming Interface (TAPI)** is middle-ware and provides a high-level programming interface for the user application. This interface takes care of handling the details of 1394 transactions across the bus, provides CSR management of some of the bus-dependent items, and provides the application layer functional usage of the hardware being provided.

The **Hardware Abstraction Layer (HAL)** is the lowest layer of firmware before the iSphynxII-based hardware. Its purpose is to provide an interface of the essential functions for the hardware. These functions isolate hardware details from the client firmware modules. This creates a stable programming environment for system-level firmware designers and a flexible environment for hardware designers. The HAL provides no 1394 bus management services and minimal event notification services to client (user-developed) firmware modules.

Note that the term *Lynx* is used throughout this document. This is a term that refers to any TI 1394 link layer controller, including the link portion of TSB43AA82. Also note that the part number TSB43AA82 and its common name, iSphynxII, are used interchangeably.

3. Architecture

3.1 System Firmware

The architecture for use of the iSphynxII LynxSoft Target Application is shown below in Figure 1. The firmware is designed in a modular fashion to help aide in the re-usability of the firmware as well as to insulate the upper layers from the details. The HAL layer provides the basic functionality of the hardware and helps to protect the upper layers from changing hardware. The API layer provides the protocol necessary to implement 1394 and SBP-2 transactions as well as managing node responsibilities. These two layers provide a simplified interface for the development of the embedded application and attempt to minimize the application's need for specific 1394 and TSB43AA82 details.

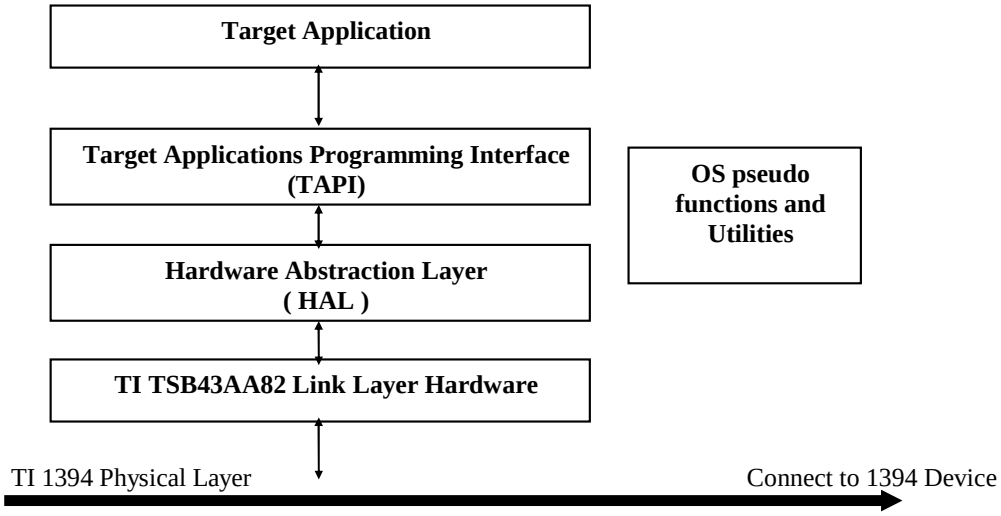


Figure 1: LynxSoft Target Firmware Architecture

3.2 System Hardware

Fig. 2 shows a hardware reference platform for which the iSphynxII firmware was designed. The platform is designed to aide the developer in understanding the interface for the TSB42AA82 as well as provide an example to design a system.

This platform is hereafter referred to simply as "the hardware".

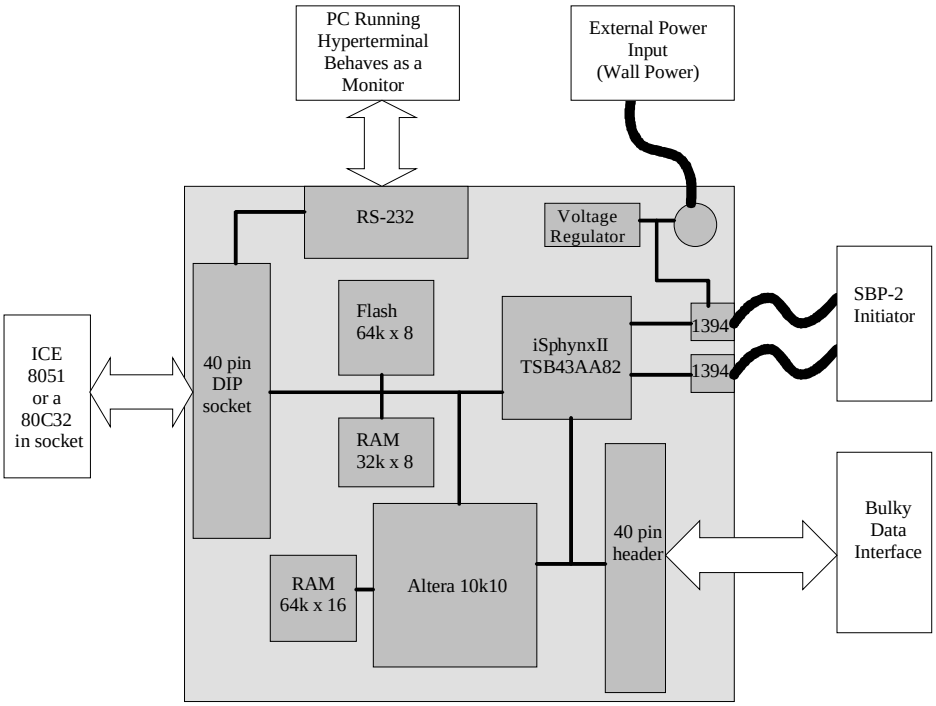


Figure 2 TSB43AA82 Hardware Reference Platform

4. Applications

4.1 Introduction

The TSB43AA82 firmware provides two sample applications that can be used to understand the capabilities of the part, the operational responsibilities of a node, as well as code to cut and paste for your own needs. A final embedded product application will use the applications as a guideline in its development. The provided applications do not handle all errors that could possibly be returned nor take error corrective action in all cases.

There are two compiler defines in the code that provide switching between the applications, depending upon which application you wish to work on. Both defines may be declared in order to create an application that has the Monitor and HDD applications concurrently in operation. This allows the user to view registers and memory and check system status while the HDD application responds to requests.

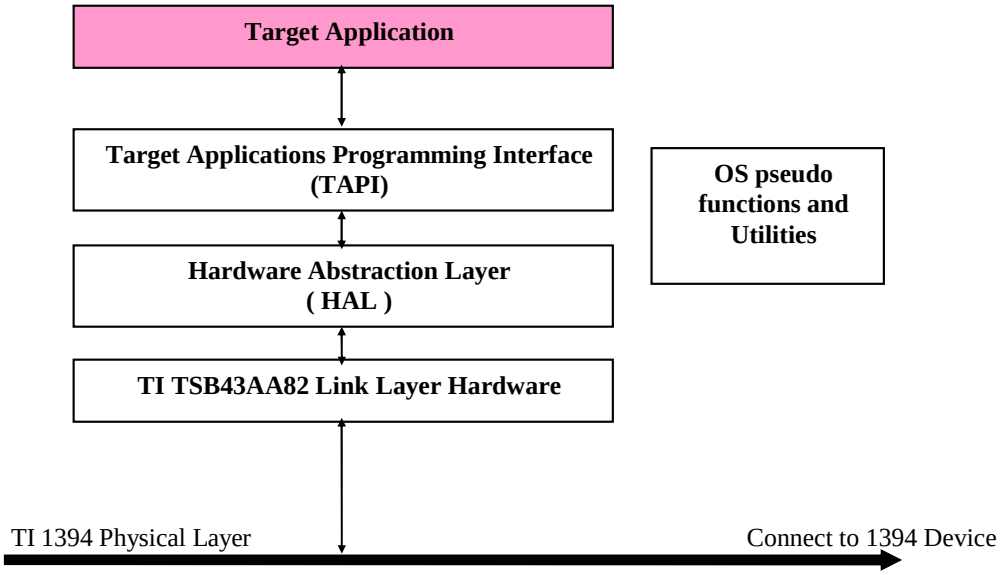


Figure 3 LynxSoft Target Firmware Architecture (App)

4.2 Monitor Application

4.2.1 Overview

This section describes the overall use of the application as well as a description of most of the application's commands. The Monitor program accesses the 1394 bus through the API and HAL layers. The following compiler defined items control the building of this application.

RS232 – define this in the project options to create a build of the Monitor application.

HOOK_CALLBACKS – define this if you want the Monitor application callbacks to be in the call process with the HDD callbacks. The Monitor callbacks write info to the main window when one of these events occur.

The Monitor application runs on top of the Target Application Programmers Interface (TAPI). This application goes around the TAPI in some instances in order to access information that is normally not needed by an application. The test application provides a way for users to utilize the TAPI interface functions and perform basic operations of the link layer controller. It is also a source of sample code demonstrating the use of the embedded Target API.

4.2.2 Operating Instructions

This application runs with the iSphynxII LynxSoft TAPI and HAL on TI's TSB43AA82 hardware reference platform. The firmware is provided for this application.

To run the application you must have a RS-232 serial port connecting the hardware to a PC. Use the Windows Monitor program HyperTerminal with the following settings:

- Bits Per Second: 9600
- Data Bits: 8
- Parity: None
- Stop Bits: 1
- Flow Control: None
- On the Settings tab, under ASCII Setup, DESELECT the box that reads "Echo typed characters locally"

The terminal display contains 2 header lines that provide information about the test app version as well as information on the state of the bus. The bottom line of the menu is where commands are entered as well as the current 1394 CycleTimer value. The central portion of the screen is used for additional prompts or information to be displayed. Pressing the key corresponding to the desired action enters a

command. Some commands, such as DMA Utility Functions, require further user input. The following sections describe the available commands.

4.2.3 Common Menu Commands

The HyperTerminal screen will always display the title and version, node number, root status, bus number, node count, bus reset count, and cycle time. These values are updated with each command. Each menu contains a list of common commands described in detail below.

4.2.3.1 Help – List of commands

The help menu can be viewed by pressing 'h' at the prompt. This menu lists all available commands. Any of these commands can be executed by pressing the corresponding letter or number. NOTE: the help menu does not have to be displayed to execute the listed commands, as long as the prompt is displayed.

4.2.3.2 Clear screen

Pressing 'c' at the prompt will clear the screen. The resulting screen will only display the title, bus reset information, and the prompt.

4.2.3.3 Bus reset

Pressing 'i' at the prompt will generate a bus reset. This command generates a bus reset and reinitializes the firmware. The resulting screen will notify the user that a bus reset has occurred. In addition, the bus reset status and any callbacks will be displayed.

4.2.3.4 Versions

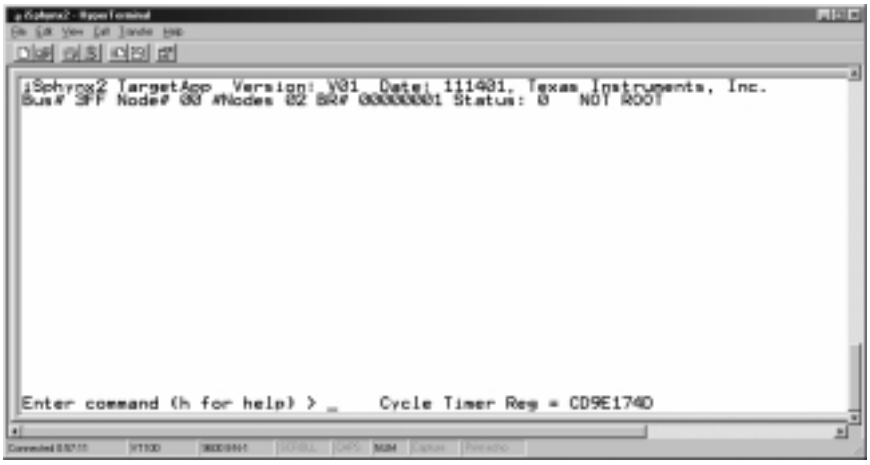
Pressing 'v' at the prompt will display the firmware and hardware versions. This command will clear the screen and display the Test App, Target API, LynxHAL, and hardware version numbers.

4.2.3.5 Exit

Pressing 'x' at the prompt will cause you to exit the current menu. This will allow you to return from a sub-menu but does not exit the "main" menu.

4.2.4 Main Window

Below shows an example screenshot of the main window for the Monitor application.

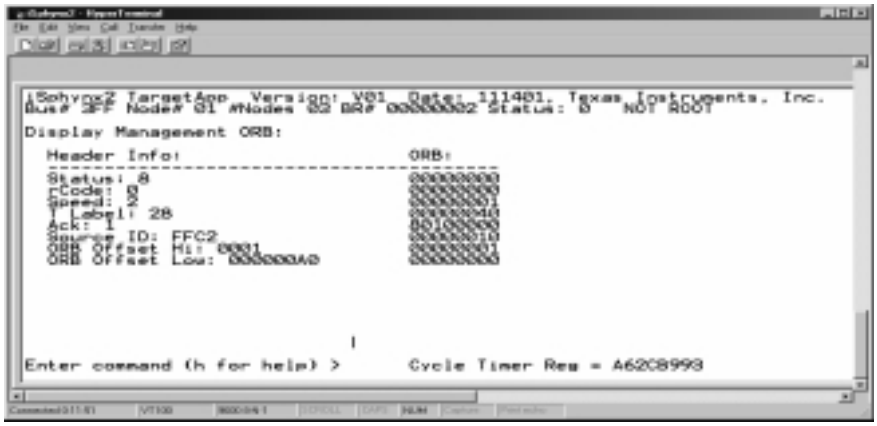


4.2.4.1 Help Menu

The main menu command "H – Help" displays a list of the available commands at this level.

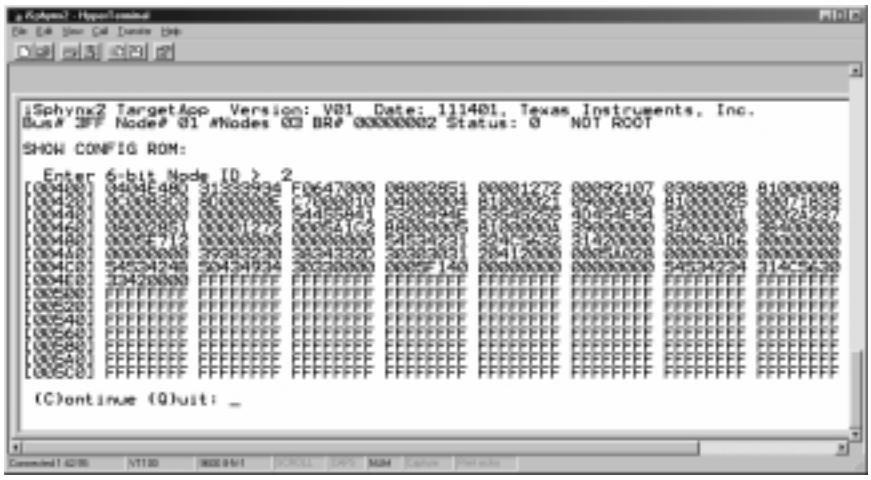
4.2.4.2 Last Management ORB

The main menu command "M – Show Last Management ORB" displays the data for the last management ORB that has been received by the firmware. It provides the ORB status, source node and address from where it was retrieved and the 8 quads of the ORB itself.



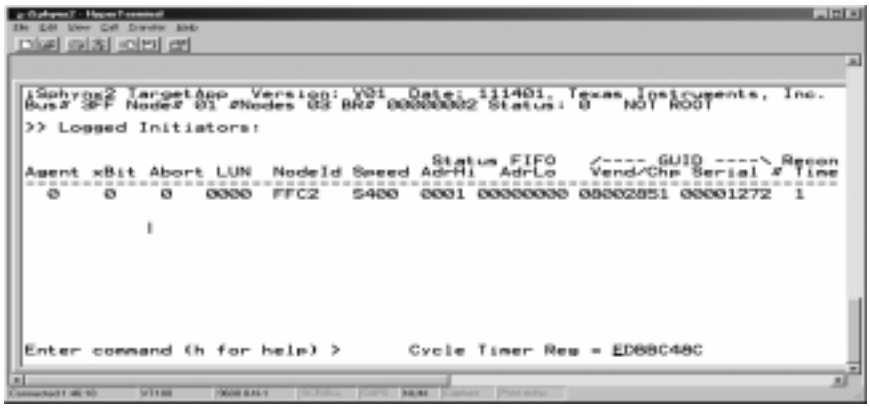
4.2.4.3 Node's Congif Rom

The main menu command "R – Show Config Rom" displays the config rom data for any node on the bus.



4.2.4.4 Show Logged Initiators

The main menu command "L – Logged Initiators" displays all currently logged in initiator node information as shown in .

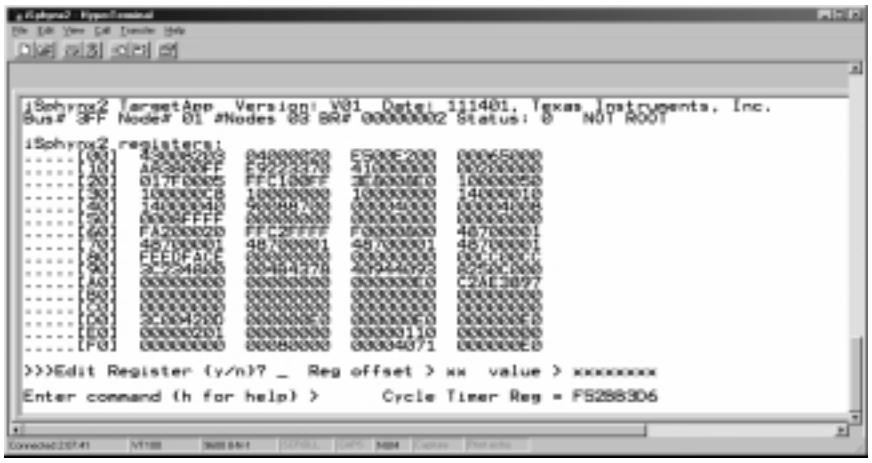


4.2.4.5 Last Unsolicited Async Pkt

The main menu command "U – Show Last Unsolicited" displays the data for the last ASYNC packet received by the firmware.

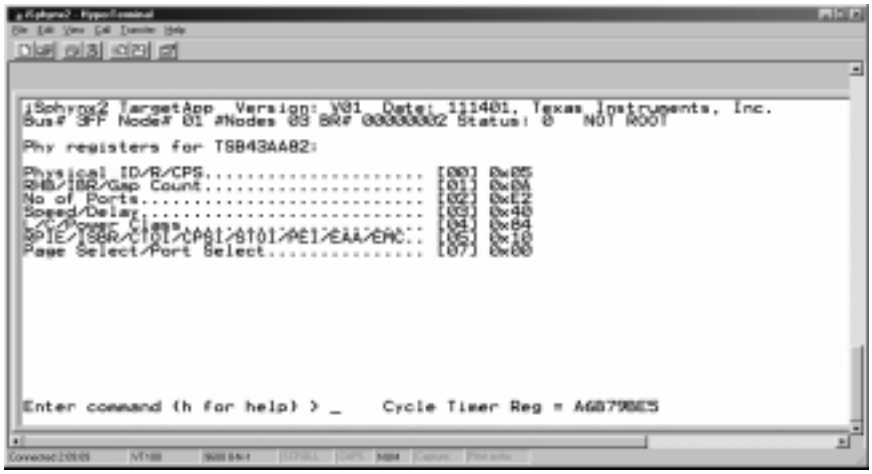
4.2.4.6 Show/Edit Lynx Registers

The main menu command "0 – Show/Edit Lynx" displays the current value of all the link's registers and then give you the capability to modify them.



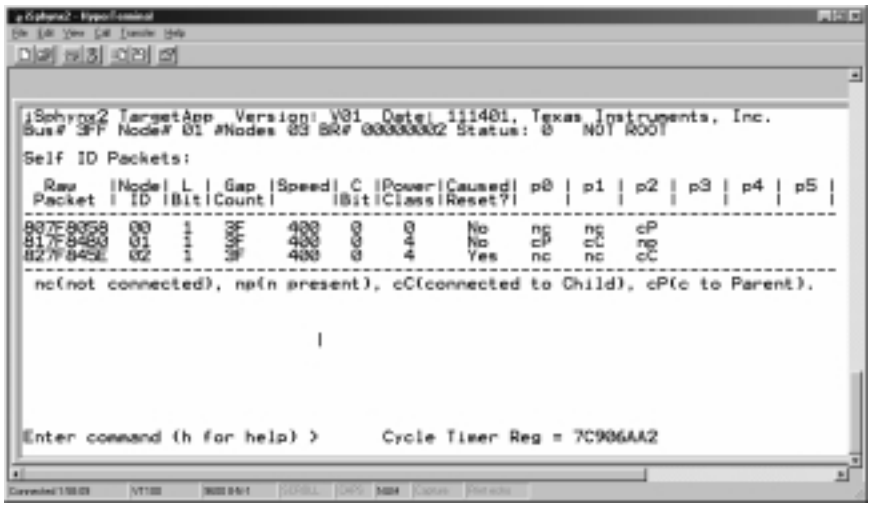
4.2.4.7 Show PHY Registers

The main menu command "1 – Show PHY Regs" displays the current PHY register values.



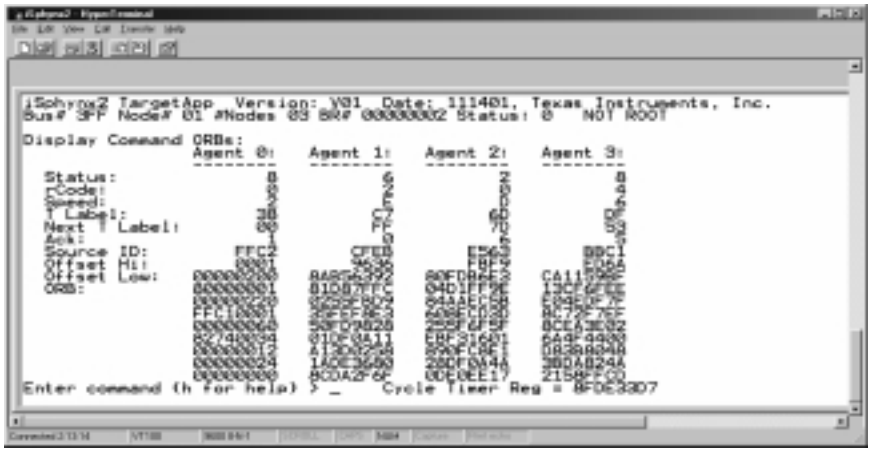
4.2.4.8 Show Received Self-IDs

The main menu command "2 – Show Received Self-IDs" displays the data for the collected self-ids as show below.



4.2.4.9 Show Received Command ORBs

The menu command "3 – Show Rx'd Cmd ORBs" displays the data for all Command Agents.

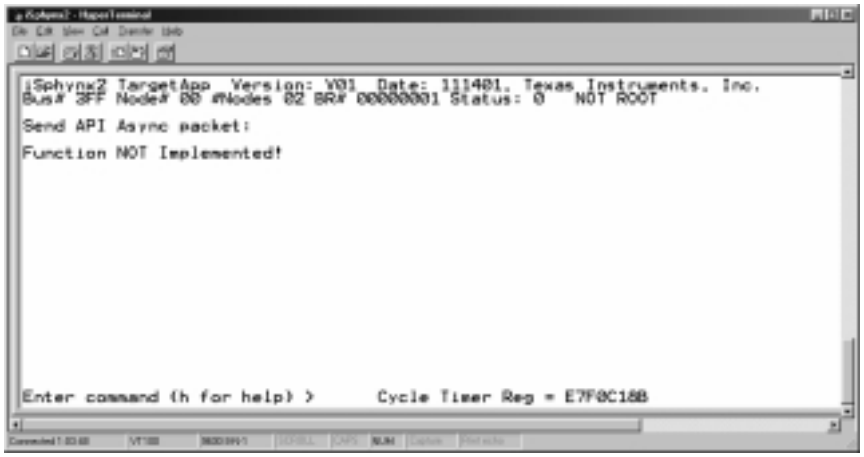


4.2.4.10 Send PHY Packet

The menu command "4 – Send PHY pkt" displays a set of prompts for the user to fill in and then send the request if the user still wishes.

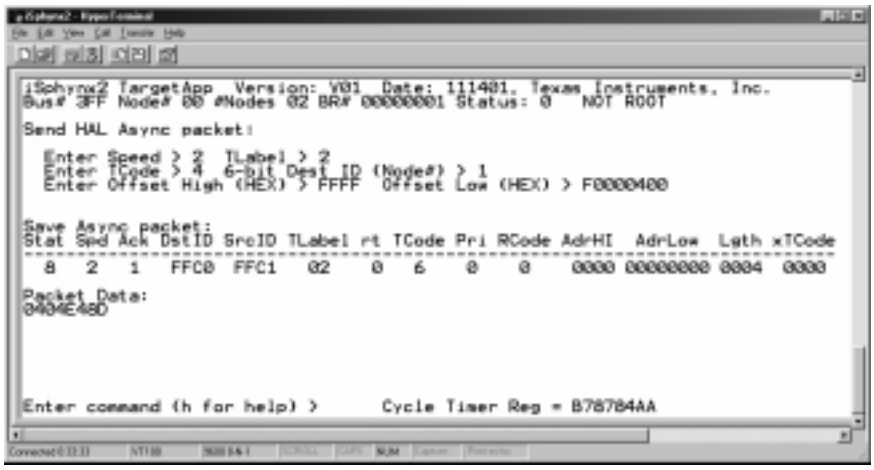
4.2.4.11 Send Async R/W Packet (API)

The menu command "5 – Send Async R/W" is NOT implemented at this time



4.2.4.12 Send Async R/W Packet (HAL)

The menu command "6 – Send Async R/W" displays a set of prompts for the user to fill in and then send the request and display the returned data if any.

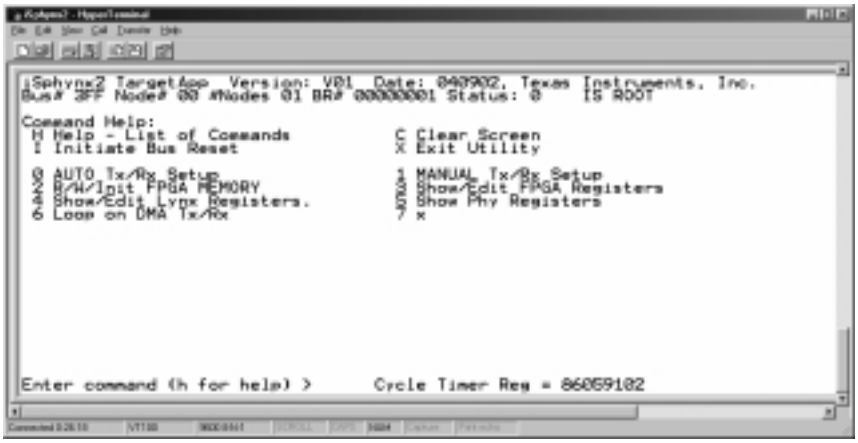


4.2.4.13 Loop Async R/W Packet (HAL)

The menu command "8 – Loop Async R/W" displays a set of prompts for the user to fill in and then send the request as many times as specified and display the returned status for each packet.

4.2.5 DMA Utilities Menu

The main menu command "7 – DMA Utility Functions" displays the menu as show below. This menu provides the operator the capability of executing DMA operations across the bus.



4.2.5.1 AUTO Tx/Rx Setup

The menu command "0 – AUTO Tx/Rx Setup" displays a set of prompts for the user to fill in and then initiate a DMA request across the bus.

4.2.5.2 MANUAL Tx/Rx Setup

The menu command "1 – MANUAL Tx/Rx Setup" is a sub-menu that provides the user various options to set up the hardware before initiating a DMA request.

4.2.5.3 R/W/Init FPGA MEMORY

The menu command "2 – R/W/Init FPGA MEMORY" displays a set of prompts for the user to fill in to read/write data into the FPGA memory or to initialize a range of memory to all 0xFF's with a quadlet at each 0x100 address being set to the address value.

4.2.5.4 Show/Edit FPGA Registers

The menu command "3 – Show/Edit FPGA Registers" will allow the user to read/write to the FPGA registers on the hardware.

4.2.5.5 Loop on DMA Tx/Rx

The menu command "6 – Loop on DMA Tx/Rx" displays a set of prompts for the user to fill in and initiate a DMA request across the bus.

4.2.6 Manual TX/RX Setup Menu

The DMA utilities menu command "1 – Manual Tx/Rx Setup" displays the menu as show below. This menu provides the operator the capability of setting up manually different aspects of a data transfer using the hardware.

4.2.6.1 Setup DMA Tx/Rx Mode

The menu command "0 – Setup DMA Tx/Rx Mode" displays a set of prompts for the user to fill in the direction of the DMA transfer as well as what operational mode.

4.2.6.2 Setup BDIF Xfer Mode

The menu command "1 – Setup BDIF Xfer Mode" displays a set of prompts for the user to fill in the BDIF mode (A-I) as well as control signals. This allows the user to select the interface between the device and the BDIF.

4.2.6.3 Setup FPGA Tx/Rx data

The menu command "2 – Setup FPGA Tx/Rx data" displays a set of prompts for the user to fill in the start address and byte count for a transfer from FPGA memory.

4.2.6.4 Setup BDIF Tx/Rx data

The menu command "3 – Setup BDIF Tx/Rx data" displays a set of prompts for the user to fill in the DMA control information for the DMA engine.

4.2.6.5 Tx/Rx thru BDIF I/F (PKT)

The menu command "4 – Tx/Rx thru BDIF I/F (PKT)" initiates the DMA operation and waits for the completion of the operation.

4.2.6.6 Tx/Rx thru BDIF (DPP)

The menu command "5 – Tx/Rx thru BDIF (DPP)" is NOT functional at this time. It would display a set of prompts for the user to fill in the DMA control information to create a DPP range for the DMA to Monitor.

4.2.6.7 Tx/Rx thru HOST I/F

The menu command "6 – Tx/Rx thru HOST I/F" displays a set of prompts for the user to fill in a description of packet size and number of packets to send across the bus. This assumes that you have DRD/DTD select disabled, however you can enable this and use the packetizer to send the data for you implementing a PIO mode of operation.

4.2.6.8 Tx/Rx thru BDIF (ATAPI)

The menu command "7 – Tx/Rx thru BDIF (ATAPI)" displays a set of prompts for the user to fill in the DMA control information. It will then place the BDIF into ATAPI mode of operation and initiate the operation.

4.2.6.9 Reset DTF/DRF Packetizer

The menu command "8 – Reset DTF/DRF Packetizer" displays a prompt for the user to indicate which packetizer is to be reset, then performs the operation.

4.2.6.10 Reset DTF/DRF FIFOs

The menu command "9 – Reset DTF/DRF FIFOs" displays a prompt for the user to indicate which FIFO is to be reset, then performs the operation.

4.2.7 Configuration ROM

The following is an example of a Configuration ROM setup for a test node. This function and data should be a part of the Application/Driver that uses the iSphynxII Lynxsoft TAPI. This information can be found in the application's "CNFIGROM.H" source file.

Microsoft's bus enumerator expects to find the "unit spec id" and the "unit sw version" in order to show this device in the device manager as well as load any drivers for the device. These two entries are required if you want Windows to recognize your device through Plug-n-Play.

```
#define CFR_CNFIGROM_SIZE  51  //# of quadlets
#define CFR_MNGTOFFSET_ADR 21  //Quadlet offset Management Register Address

unsigned long ulDriverConfigROM[] =
{
// BUS INFO BLOCK (required)
//0x04xxFFFF,  // 0x400 HDR: info_lgt/crc_lgt/CRC
0x0400FFFF | ((ULONG)CFR_CNFIGROM_SIZE << 16),
0x31333934,  // 0x404 BUS Identifier
0x00FFA212,  // 0x408 Node Capabilities
0x08002800,  // 0x40C vendor id/chip id hi
0xf12421FF,  // 0x410 chip id lo
//
// DIR: ROOT DIRECTORY (required)
0x0007FFFF,  // 0x414 HDR: crc_lgt/CRC
0x0C0083C0,  // 0x418 node capabilities
0x03080028,  // 0x41C module vendor id
0x81000015,  // 0x420 text leaf #1
0x04000001,  // 0x424 Module HW Version
0x81000024,  // 0x428 text leaf #3
0x8D00000F,  // 0x42C Offset to NODE UNIQUE ID
0xD1000001,  // 0x430 unit directory offset
//
// DIR:  UNIT DIRECTORY (required)
0x000CFFFF,  // 0x434 HDR: crc_lgt/CRC
0x1200609E,  // 0x438 unit spec id
0x13010483,  // 0x43C unit sw version
0x3A000A08,  // 0x440 unit characteristics
0x3800609E,  // 0x444 command set spec id (T10)
0x390104D8,  // 0x448 command set (RBC)
0x3B0104D8,  // 0x44C command set revision
0x3C000000,  // 0x450 Firmware revision
0x54004000,  // 0x454 Mngt Agent csr offset
0x3D000001,  // 0x458 Reconnect time
0x14000000,  // 0x45C device type/lun
0x1743AA82,  // 0x460 model id
0x8100000C,  // 0x464 text leaf #2
//
// Leaf: Node_Unique_Id */
0x0002FFFF,  // 0x468 HDR: crc_lgt/CRC
0x08002800,  // 0x46C vendor id/chip id hi
0xf12421FF,  // 0x470 SerialNo
//
```

```
// TEXT Leaf #1: Module Vendor ID
0x0007FFFF,  // 0x474 HDR: crc_lgt/CRC
0x00000000,  // 0x478
0x00000000,  // 0x47C
0x54455841,  // 0x480 TEXA
0x5320494E,  // 0x484 S IN
0x53545255,  // 0x488 STRU
0x4D454E54,  // 0x48C MENT
0x53000000,  // 0x490 S
//
// TEXT Leaf #2:  Model ID
0x0008FFFF,  // 0x494 HDR: crc_lgt/CRC
0x00000000,  // 0x498
0x00000000,  // 0x49C
0x54534234,  // 0x4A0 "TSB4"
0x33414138,  // 0x4A4 "3AA8"
0x32204556,  // 0x4A8 "2 EV"
0x4D202841,  // 0x4AC "M (A"
0x50502920,  // 0x4B0 "PP) "
0x00000000,  // 0x4B4
//
// TEXT Leaf #3:  HW Version
0x0004FFFF,  // 0x4B8 HDR: crc_lgt/CRC
0x00000000,  // 0x4BC
0x00000000,  // 0x4C0
0x52455620,  // 0x4C4 "REV "
0x302E3100   // 0x4C8 "0.1"

};

/* End of Configuration ROM */
```

4.3 HDD Application

4.3.1 Overview

This application provides the very basic functionality of a 1394 hard disk drive using SCSI RBC. It uses either on-board memory for a virtual drive or a real HDD if one is connected. The following compiler defined items control the building of this application.

SCSI_TARGET – define this in the project options to create a build of the HDD application.

4.3.2 Operating Instructions

If using the specified hardware reference platform as a base for the application, the node will look like an unpartitioned/unformatted HDD to Windows. Because the Windows operating system requires a minimum of 2MB for the FAT information, the node will only show up in the Device Manager. The platform's limited memory prohibits using FDISK to set up partitions as well as formatting to make it available to use.

If the right hardware is available and a real HDD is being used as a base for the application, there are several files provided that are not tied into the project that operate on the ATA interface. Some modifications will need to be made based on compiler and platform in use.

4.3.3 Configuration ROM

The following is an example of a Configuration ROM setup for a HDD node. This function and data should be a part of the Application/Driver that uses the iSphynxII LynxAPI. This information can be found in the file "SCSI_HDD.H".

```
#define CFR_CNFIGROM_SIZE  51  //# of quadlets
#define CFR_MNGTOFFSET_ADR 21  //Quadlet offset Management Register Address

unsigned long ulDriverConfigROM[] =
{
// BUS INFO BLOCK (required)
//0x04xxFFFF,  // 0x400 HDR: info_lgt/crc_lgt/CRC
0x0400FFFF | ((ULONG)CFR_CNFIGROM_SIZE << 16),
0x31333934,  // 0x404 BUS Identifier
0x00FFA212,  // 0x408 Node Capabilities
0x08002800,  // 0x40C vendor id/chip id hi
0xf12421FF,  // 0x410 chip id lo
//
// DIR: ROOT DIRECTORY (required)
0x0007FFFF,  // 0x414 HDR: crc_lgt/CRC
0x0C0083C0,  // 0x418 node capabilities
0x03080028,  // 0x41C module vendor id
0x81000015,  // 0x420 text leaf #1
0x04000001,  // 0x424 Module HW Version
0x81000024,  // 0x428 text leaf #3
```

```
0x8D00000F,  // 0x42C Offset to NODE UNIQUE ID
0xD1000001,  // 0x430 unit directory offset
//
// DIR:  UNIT DIRECTORY (required)
0x000CFFFF,  // 0x434 HDR: crc_lgt/CRC
0x1200609E,  // 0x438 unit spec id
0x13010483,  // 0x43C unit sw version
0x3A001E08,  // 0x440 unit characteristics
0x3800609E,  // 0x444 command set spec id (T10)
0x390104D8,  // 0x448 command set (RBC)
0x3B0104D8,  // 0x44C command set revision
0x3C000000,  // 0x450 Firmware revision
0x54004000,  // 0x454 Mngt Agent csr offset
0x3D000001,  // 0x458 Reconnect time
0x14000000,  // 0x45C device type/lun
0x1743AA82,  // 0x460 model id
0x8100000C,  // 0x464 text leaf #2
//
// Leaf: Node_Unique_Id */
0x0002FFFF,  // 0x468 HDR: crc_lgt/CRC
0x08002800,  // 0x46C vendor id/chip id hi
0xf12421FF,  // 0x470 SerialNo
//
// TEXT Leaf #1: Module Vendor ID
0x0007FFFF,  // 0x474 HDR: crc_lgt/CRC
0x00000000,  // 0x478
0x00000000,  // 0x47C
0x54455841,  // 0x480 TEXA
0x5320494E,  // 0x484 S IN
0x53545255,  // 0x488 STRU
0x4D454E54,  // 0x48C MENT
0x53000000,  // 0x490 S
//
// TEXT Leaf #2:  Model ID
0x0008FFFF,  // 0x494 HDR: crc_lgt/CRC
0x00000000,  // 0x498
0x00000000,  // 0x49C
0x54492056,  // 0x4A0 TI V
0x49525455,  // 0x4A4 IRTU
0x414C2044,  // 0x4A8 AL D
0x49534B20,  // 0x4AC ISK
0x44524956,  // 0x4B0 DRIV
0x452E0000,  // 0x4B4 E.
//
// TEXT Leaf #3:  HW Version
0x0004FFFF,  // 0x4B8 HDR: crc_lgt/CRC
0x00000000,  // 0x4BC
0x00000000,  // 0x4C0
0x52455620,  // 0x4C4 "REV "
0x302E3100   // 0x4C8 "0.1"
};
```

5. System Utilities

5.1 Introduction

This section of the document describes the psuedo OS utilities that are provided with the firmware for use on the TSB43AA82 hardware reference platform.

5.2 RS-232 Functions

Several utility functions are provided that allow the application program to utilize the RS-232 port to provide interaction with a user. These functions provide for transmit and receive of character data as well as numeric-to-character and character-to-numeric conversions.

5.3 Call Queue

This psuedo OS feature provides the application the ability to perform "tasking" type operations. The API and HAL layers utilize this feature in order to provide "events" to the upper layers. The Call Queue is nothing more than an array of functions to be called that have been placed into the queue. The functions are called in the order that they are put in. The application must provide a point in its operation to "YieldToCallQue". This allows those functions placed into the queue to be processed and the queue emptied.

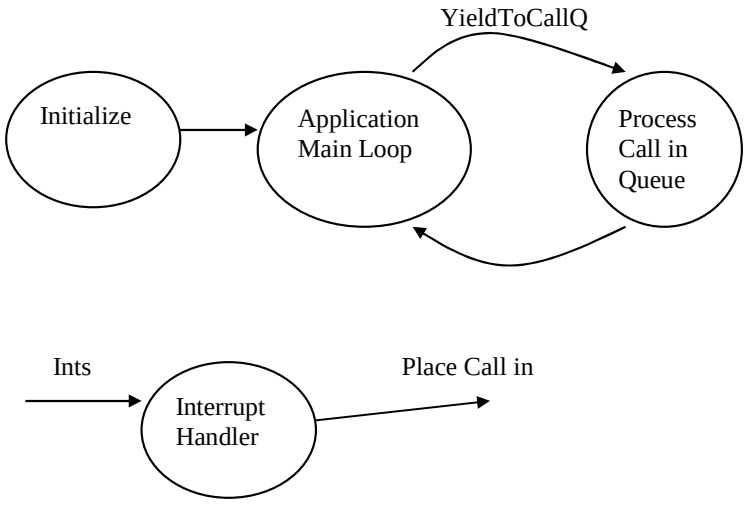


Figure 4 Call Queue Operation

6. Applications Programming Interface (API)

6.1 Introduction

This section of the document describes TI's implementation of the **iSphynxII LynxSoft Target Application Programming Interface** (TAPI) for the TSB43AA82 Link Layer Controller. The goal of this API is to provide a transportable library of functions that can be utilized by an embedded application and is intended to help isolate the developer from IEEE 1394, SBP-2 command details as well as hardware-specific knowledge.

The available functions provide the initialization, setup, and the transport of Asynchronous data as well as support for the SBP-2 command protocol. This leaves the developer to understand the concepts and capabilities of the 1394 bus as well as the overall SBP-2 command protocol.

The developer will also need to implement the device drivers for the different types of devices that the developer expects to interface with via the 1394 bus. This API does not provide mechanisms to enable cycle master, isochronous resource manager (IRM), or bus manager (BM) capability.

This section of the document provides an overview of the interface (defines, callbacks, structures, and prototypes) that the API provides to the upper layers. The architecture of the expected use for the LynxSoft API is shown in the diagram below.

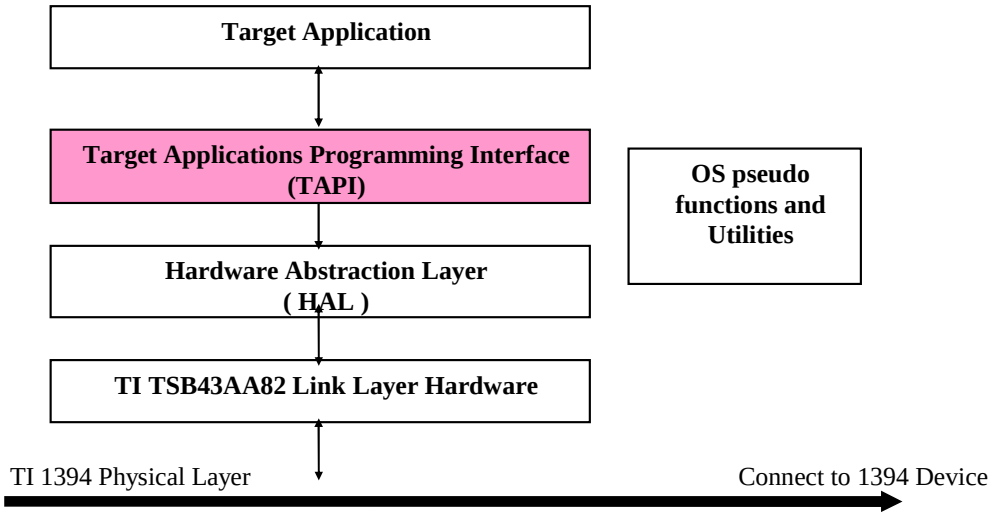


Figure 5 LynxSoft Target Firmware Architecture (API view)

6.2 Target API Theory of Operation

6.2.1 Target API Description

The iSphynxII LynxSoft Target API provides the necessary functions needed to do IEEE 1394 bus operations to request information from other nodes or respond to queries from other nodes. The use of callbacks allow the application to respond to queries as well as handle data completion events.

6.2.2 Initialization

The iSphynxII LynxSoft Target API cannot be utilized until it has been initialized. The application must supply information needed in order to setup the TSB43AA82 as well as have the hooks to callback the application to notify it of certain events. The application must supply the Target API the data for the config rom, and the API will calculate the CRCs for the rom. Also, the application must initialize any information it needs before 1394 operations are enabled.

6.2.3 Asynchronous Sequence

The following Asynchronous sequence needs to be followed for successful operation. The table below describes some asynchronous functions and some of the required sequence considerations before using the functions.

Operation	Result
Application dependant	Prior to invoking an async request, the application would allocate memory for any data in the transfer, fill in the input structure with the appropriate information.
sbiAsyncWrite, sbiAsyncRead	The application firmware can read or write to any 1394 address space that has been made available by the target device.
Application dependant	Process the data returned by a request or free the allocated memory

6.2.4 SBP-2 Command Sequence

Operation	Result
Process Command	When Command ORB arrives, determine what the command is and perform needed action. Set appropriate error information.
SphnxAPISetCmdDepStatus & SphnxAPICmdTerminate	When command processing is complete, generate SBP-2 status information and send status response to requestor.

6.2.5 DMA Sequence

The following DMA sequence needs to be followed for successful operation. The table below describes some DMA functions and some of the required sequence considerations before using the functions.

Operation	Result
Device Preparation	Issue commands to the device hardware that will place it into a state to transfer or receive data if using the BDIF.
SphnxAPIDMARWSetup	Provide the information to describe the amount of data, any packet size limitations and whether it will use the BDIF or PIO mode for moving the data in/out of FIFOs. The hardware will be setup to perform the requested operation but not started.
SphnxAPIDMARWExec	When the application is ready for the data, this function will start the DMA hardware that has been setup.
Transfer Data	If using HOST/PIO modes for data, then move the data from/to the FIFOs to/from the device.
Transfer Complete	Callback function will be invoked to allow the application to complete operations on the device or SBP-2 protocol and error checking.

6.3 Defines

The following table describes some of the "C" defines used in the TSB43AA82 API interface. Note: this does not describe all defines used in the API, only ones that are relevant to the upper layers.

Define	**Description**
MAX_LOGICAL_UNIT	Maximum number of logical units to be supported.
SBP2_STATUSOK	Baseline SBP-2 status for command status
SPEED100	Used in parameter for Async request to set the speed of the transaction on the bus.
SPEED200	Used in parameter for Async request to set the speed of the transaction on the bus.
SPEED400	Used in parameter for Async request to set the speed of the transaction on the bus.
SPEEDFASTEST	Used in parameter for Async request to use the maximum speed of target node for transaction.
USE_AUTOPAUSE	Used if need to restrict flow of data through DRF or requests to a remote node.

6.4 Structures

There are several structures utilized in the TSB43AA82 API to provide the application layer the knowledge of what information is needed to perform a function. Some structures are used to pass information into the API while others are supplied by the API as parameter information in callbacks.

6.4.1 CMDMNGTDATA

The following is the format of the API's CMDMNGTDATA which is a sub-structure defined within the SBP-2 spec on information contained within the command ORB. Due to the usage of bit fields within the structure, you should verify that your compiler places them in correct order.

```
typedef struct _CmdMngtDta {
  UINT PageSize   : 3;
  UINT PageEnable : 1;
  UINT MaxPayLoad : 4;
  UINT Speed      : 3;
  UINT Direct     : 1;
  UINT Rsvrd1     : 1;
  UINT Rq_FMT     : 2;
  UINT Notify     : 1;
} CMDMNGTDATA, *PCMDMNGTDATA;
```

All the following fields are defined as part of a command ORB in the SBP-2 specification.

- **PageSize** is the underlying page size of memory being accessed.
- **PageEnable** will indicate if a page table is being used.
- **MaxPayLoad** is the maximum data transfer request.
- **Speed** is the desired transfer speed for the data.
- **Direct** is the direction of data transfer.
- **Rq_FMT** is the ORB format defined in the SBP-2 specification.
- **Notify** will indicate whether the SBP-2 status block is required.

6.4.2 ADDRPOINTER

The following is the format of the API's ADDRPOINTER which is a sub-structure defined within the SBP-2 spec on information contained within the command ORB.

```
typedef struct _AddressPtr {
  USHORT  NodeID;
  USHORT  OffsetHi;
  ULONG   OffsetLo;
} ADDRPOINTER, *PADDRPOINTER;
```

6.4.3 ORBPOINTER

The following is the format of the API's ORBPOINTER which is a sub-structure defined within the SBP-2 spec on information contained within the command ORB.

```
typedef struct _SBP2OrbPtr {
  USHORT  n_rsvd;
  USHORT  OffsetHi;
  ULONG   OffsetLo;
} ORBPOINTER, *PORBPOINTER;
```

6.4.4 SBP2CMND_ORB

The following is the format of the API's SBP2CMD_ORB which is defined within the SBP-2 spec and is used in the command callback routine to provide information on the command being sent to the node.

```
typedef struct _SBP2CmndOrb {
  ORBPOINTER  NextOrb;
  ADDRPOINTER DataDesc;
  USHORT MngtData;
  USHORT DataSize;
  ULONG CmdBlock[COMMAND_BLOCK_SIZE_QUADS];
 } SBP2CMND_ORB, *PSBP2CMND_ORB;
```

6.4.5 ASYNC_REQPKT

The following is the format of the API's ASYNC_REQPKT. This structure is used during the indication callback routine to provide the data from the request to the application. This structure is allocated dynamically in order to be able to adjust to variable size block packets. Data for a block type packet will be stored starting at the block_data field.

```
typedef struct {
  ULONG  ulGenerationCnt;  // To check for BR, to prevent stale pkts
  ULONG  ulCycleTime;  // CycleTimer value at time pkt read from FIFO
  struct {     /* first quadlet */
    UCHAR  Tcode_Pri; //Tcode upper 4 bits, Pri lower 4 bits
    UCHAR  TLabel_rt; //TLabel upper 6 bits, rt lower 2 bits
    USHORT dst_id;    //Upper 16 bits of quad
  } Q1; /* struct 1st quad */

  struct {      /* second quadlet */
    USHORT dstHi_Rcode;  //RCode is upper 4 bits of 16 bit field
    USHORT src_id;       //Upper 16 bits of quad
  } Q2; /* union 2nd Quad */

  ULONG    dstLo;  /* 3rd Quad */
```

```
  union {      /* fourth quadlet */
    ULONG    quad_data;
    struct {
      USHORT   extended_tcode;
      USHORT   data_length;
    } x; /* struct */
  } Q4; /* union 4th Quad */

  ULONG    block_data[1];      /* N quadlets of block data */

} ASYNC_REQPKT, *PASYNC_REQPKT;
```

- **ulGenerationCnt** is used to check for busreset to prevent stale packets from going out on bus.
- **ulCycleTime** is the CycleTimer value at time packet read from FIFO.
- **Q1 – Q4** are basically the header quadlets of either a block or quadlet ASYNC packet.
- **block_data** is used to provide a C reference to an array for dynamically growing memory to contain the data that arrives with a block packet.

6.4.6 SBI_INIT_INFO

The following is the format of the API's initialization structure along with associated type definitions. NOTE: The following typedefs supply the format for CALLBACKS to the APPLICATION space by the API.

```
typedef void (*PAPICB)( void* pContext );
typedef void (*PAPIBRTCB)( PBUSRESETINFO pInfo );
typedef void (*PAPICMDCB)( PSBP2CMND_ORB pCmdORB );
typedef BOOL (*PAPIPSWCB)( PUCHAR pPsw );
typedef void (*PAPIINDCB)( PASYNC_REQPKT pPkt );

typedef struct {
  CFGROMINIT  CRInfo;
  PAPIBRTCB   BusResetCB;
  PAPICB      LUNResetCB;
  PAPICB      TargResetCB;
  PAPICMDCB   CommandCB;
  PAPIINDCB   IndicationCB;
  PAPIPSWCB   VerifyPswCB;
  PAPIPSWCB   UpdatePswCB;
} SBI_INIT_INFO, *PSBI_INIT_INFO;
```

- **CRInfo** is the structure containing Config Rom information needed to setup device.
- **BusResetCB** is the address of the callback routine for BusReset processing.
- **LUNResetCB** is the address of the callback routine for LogicalUnit Reset processing.
- **TargResetCB** is the address of the callback routine for Target Reset processing.
- **CommandCB** is the address of the callback routine for Command ORBs received.
- **IndicationCB** is the address of the callback routine for received ASYNC packets.

- **VerifyPswCB** is the address of the callback routine to verify the master or current password.
- **UpdatePswCB** is the address of the callback routine to update the current password.

6.4.7 ASYNC_RWPKT

The following is the format of the API's ASYNC_RWPKT structure which is used by the application to send read/write requests to other nodes or to send a response back to a request made by another node on the bus.

```
typedef struct
{
  UCHAR   ucSpeed;
  UCHAR   ucTlabel;
  UCHAR   ucRsv1;
  UCHAR   ucRCode;
  USHORT  usDataLength;
  USHORT  usRsv1;
  USHORT  usDestID;
  USHORT  usOffsetHi;
  ULONG   ulOffsetLo;
  ULONG   ulGenerationCnt;
  ULONG   *ulData;
} ASYNC_RWPKT, *PASYNC_RWPKT;
```

- **ucSpeed** is the 2-bit speed field for the speed at which to send the packet.
- **ucTlabel** is the 6-bit Transaction label to be associated with this packet.
- **ucRCode** is the Response Code value when sending a RESPONSE type packet, set INVALID if a request.
- **usDataLength** is the 16-bit length of the data field in bytes.
- **usDestID** is the remote node's 16-bit Destination ID, both bus and node.
- **usOffsetHi** is the remote node's 16-bit Destination offset hi.
- **ulOffsetLo** is the remote node's 32-bit Destination offset low.
- **ulGenerationCnt** is used to check for busreset to prevent stale packets from going out on bus.
- **UlData** is the pointer to array of data that will be sent as part of the packet or where data will be stored when the response is received.

6.4.8 DMA_CB_DATA

The following is the format of the API's DMA_CB_DATA structure, which is used by the application to determine if an error occurred in a DMA transaction and information that may be needed by the callback routine to send down to the API to complete an SBP-2 operation. The overall DMAStatus is composed of bit information from 3 registers, the Transaction Timer Control (0x60), DxF Acknowledge (0xA0), and DxF Control (0xB0 or 0xC0). The defines for the DMAStatus field allow parsing of the status to determine if an error occurred. DxFPktReg_? defines are provided to help analyze the Packetizer Status field and determine the Error, see DTF/DRF Packetizer Status as defined

in the data manual. Note that the status is two 16 bit fields identically defined, the upper 16 apply to the DATA packet transaction and the lower 16 apply to PAGETABLE transaction. It should be noted that when using the packetizer and processing each packet as it is received in the Data Receive FIFO (DRF), the lower 16 bits of the status field contain the number of quadlets available in the DRF.

Defines for the DMAStatus field (defined in LynxDef.h):

```
#define DxF_AckVal(DxFStat) ((DxFStat&0x0001)   )
#define DxF_LsbRsv(DxFStat) ((DxFStat&0x000E)>>1)
#define DxF_AckRcv(DxFStat) ((DxFStat&0x00F0)>>4)
#define DxF_AckErr(DxFStat) ((DxFStat&0x0100)>>8)
#define DxF_PktCtl(DxFStat) ((DxFStat&0x0600)>>9)
#define DxF_MsbRsv(DxFStat) ((DxFStat&0x7800)>>11)
#define DxF_DmaErr(DxFStat) ((DxFStat&0x8000)>>15)
```

Defines for the PKTStatus field (defined in SphnxApi.h):

```
#define DxFPktReg_Stat(PktStat)   ((PktStat>>12)&0xF)
#define DxFPktReg_Resp(PktStat)   ((PktStat>>8)&0xF)
#define DxFPktReg_Rsvd(PktStat)   ((PktStat>>5)&7)
#define DxFPktReg_AckErr(PktStat) ((PktStat>>4)&1)
#define DxFPktReg_PktAck(PktStat)  (PktStat&0xF)

typedef struct _DMAcbstruct {
  USHORT DMAStatus ;
  ULONG PKTStatus;
  PVOID pData;
} DMA_CB_DATA, *PDMA_CB_DATA;
```

- **DMAStatus** is the status of the DMA operation. It is a composite of the DxErr bit from the transaction timer control register (0x60), the two DxF Control bits (0xB0, 0xC0), and the 16 bit associated status from the DxF Acknowlede register (0xA0). Use the defines to get desired information.
- **PKTStatus** is the packetizer status as defined by register 0xE8 in data manual.
- **PData** is the pointer to either data buffer supplied or the command ORB for the operation

6.4.9 DMA_REQUEST

The following is the format of the API's DMA_REQUEST structure which is used by the application to start a DMA transaction using the hardware supplied FIFOs and possibly the hardware's packetizing features.

```
typedef void (CODEPTR PFNDMACB)(PDMA_CB_DATA);

typedef struct _DMAreqstruct {
  UCHAR    Mode;
  UCHAR    NotifyEnab;
```

```
  UCHAR    Speed;
  UCHAR    MaxPayld;
  USHORT   DestID;
  USHORT   OffsHi;
  ULONG    OffsLo;
  USHORT   Status;
  USHORT   BlockSize;
  ULONG    BlockCount;
  ULONG    DataSize;
  ULONG    ulGenerationCnt;
  PFNDMACB pHostPktCB;
  PFNDMACB pCallBack;
  PVOID    pData;
} DMA_REQUEST, *PDMA_REQUEST;
```

- **Mode** is the mode in which the DMA will operate. Host mode implies that packets will be read or written through the HOST FIFO access registers. This is basically manual loading of a packet or reading it out with no usage of the packetizer engine. The mode PIOxxx implies manual reading/writing of packets used in conjunction with the packetizer engine. The mode BDIxxx implies that the Bulky Data Interface will be used for reading/writing of data for the packets through the usage of the packetizer engine.
- **NotifyEnab** is currently unimplemented. It would be used to setup the auto-response SBP-2 status for the DMA completion.
- **Speed** is the desired speed at which to transmit a packet.
- **MaxPayld** is used to set the overall packet size used by packetizer. This field is based on the max_rec field encoding used in the 1394 specification. If the field is zero, the API will use a default value of 7 or if greater than zero the following applies 2^(n+2) packet sizes (n=7 implies pkt of 512 bytes). If a command ORB is being used to create the DMA operation, the supplied command ORB's adjusted Maxpayload field will be checked and whichever is smallest will be used.
- **DestID** is the 16 bit bus and node IDs of the remote node to which the data is being sent.
- **OffsHi** is the 16 bit 1394 offset high address of the remote node to which the data is being sent.
- **OffsLo** is the 32 bit 1394 offset low address of the remote node to which the data is being sent.
- **Status** is the status of the setup operation. If this field is non-zero upon return from the setup call, then an error has occurred and the DMA operation can not be completed.
- **BlockSize** is a multi-purpose field. If zero, then the API will packetize the data based on the input MaxPayld field, ORB Maxpayload field or if the destination ID is to a logged-in initiation, the max_rec value from config rom. If greater than zero, this field can specify the packet payload size, the overall data size using the MaxPayld field to provide packetization, or if the data is 64K or larger it becomes the blockcount designation.
- **BlockCount** is a multi-purpose field. It is normally used to designate the number of blocks of data that need to be packetized and used in conjunction with BlockSize to designate the total data transfer size. If for some reason the blocksize is greater than 64K, then this field is used to represent the blocksize and BlockSize becomes the BlockCount field.

- **DataSize** is the total data transfer size. This is used when the API uses default packetization of the data.
- **ulGenerationCnt** is used to check for busreset to prevent stale packets from going out on bus.
- **pHostPktCB** is a pointer to the application's callback function when a packet is received into the DRF and DMA mode is HOST or PIO.
- **pCallBack** is a pointer to the application's callback when the DMA packetizer has completed the data transfer that was requested.
- **pData** is a pointer to the command ORB from which to pull information on how to transfer the data. Set this field NULL if no command ORB associated with this transfer.

6.5 Callbacks

There are several types of callbacks supported in the TSB43AA82 API. These callbacks are provided by the application layer using the structures defined. If the upper layers do not wish to implement a certain callback, a NULL pointer can be passed into the API and the callback will not be executed. The following sections describe these callbacks in detail.

6.5.1 Bus Reset Callback

This callback is executed when a bus reset occurs and a pointer to the bus information as defined by the global structure BUSRESETINFO will be passed to the function. This function will be called after the API has performed its needed processing. The API tries to limit the calls back to the application on a multi-busreset occurrence. This is done at the hardware level in that only when Self-IDs are collected is the API notified and the API attempts to shut down calls to the application if another reset occurs while the API is processing a busreset. The application will need to possibly clear or reset the device due to all dataflow being shut down as well as reset global information that may be needed.

6.5.2 Indication Callback

This callback is executed when a packet is received from a remote node requesting a 1394 transaction of some kind. The API will filter out requests addressed to the 1394 CSR and Config ROM address space (0xFFFF 0xF0000000 through 0xFFFF 0xF00003FC). It is the application's responsibility to virtualize the rest of the 1394 address space. This would be used to possibly provide access to special or private device registers or a special data/command protocol.

6.5.3 Reset Callbacks

These callbacks are executed when a Management ORB request for LUN or Target reset is received. This allows the application to reset and shut down any ongoing operations or re-initialize the state of the device and firmware.

6.5.4 Password Callbacks

These callbacks are executed during the Management ORB request for Login or Update Password. There are 2 callback functions, one to perform verification of the supplied password the other to provide an update of the password.

6.5.5 Command ORB Callback

The command ORB callback will be executed each time a command ORB is received by the API and the application is sent a pointer to where the Command ORB is stored as specified in the SBP-2 spec. This pointer is then used in subsequent calls to the API to provide information needed by the API to complete SBP-2 type transactions. It is the application's responsibility to parse the command information and determine the actions required as the API does not know what command set will be used by the device and just oversees the SBP-2 transport protocol.

6.5.6 Data End Callback

This callback is called when a DMA operation has been completed. This callback receives a pointer to a DMA_CB_DATA containing the overall status of the DMA transaction. This allows the application to perform needed error detection and correction. These callbacks also allow for the application to start other processing that might be needed.

6.6 Basic Prototypes

6.6.1 sbiPresentLynxVersion

Description: This function returns the version information defined within the iSphynxII LynxSoft firmware. It provides both the API and HAL layers' version information as well as the hardware's vendor id and revision.

Inputs: None

Outputs: pqAPIswVersion – The API's version information (rrmmddyy) rev, month, day, yy in binary coded decimal.
pqHALswVersion – The HAL's version information (rrmmddyy) rev, month, day, yy in binary coded decimal.
pqHardwareVersion – The hardware's version information (vvvvvvrr), where v is the Vendor ID and r is the hardware revision.

Prototype:

```
void sbiPresentLynxVersion (ULONG*      pqAPIswVersion,
                            ULONG*      pqHALswVersion,
                            ULONG*      pqHardwareVersion   );
```

6.6.2 sbiInitialize

Description: This function initializes the iSphynxII LynxSoft API and lower level HAL layers. This is required before any other function is accessed.

Inputs: pInit – Pointer to structure containing needed information to setup node.

Outputs: RtnVal – SBI_NO_ERROR (zero) signifies successful completion of this function. A non-zero return indicates error condition. See error code list in Appendix A.

Prototype:

```
int sbiInitialize( PSBI_INIT_INFO pInit );
```

6.7 1394 Prototypes

6.7.1 sbiAsyncRead

Description: This function provides the application the ability to perform 1394 asynchronous read requests to other nodes on the bus using the API firmware. The information provided by the application layer is used to generate the read request on the bus and the function will return to the application layer when the transaction has completed or an error occurs.

Inputs: psReqPkt – Pointer to the request structure containing the information needed to create and complete a 1394 async transaction.

Outputs: RtnVal – SBI_NO_ERROR (zero) signifies successful completion of this function. A non-zero return indicates error condition. See error code list in Appendix A.

Prototype:

USHORT sbiAsyncRead(PASYNC_RWPKT psReqPkt);

6.7.2 sbiAsyncWrite

Description: This function provides the application the ability to perform 1394 asynchronous write requests and responses to other nodes on the bus using the API firmware. The information provided by the application layer is used to generate the write request on the bus and the function will return to the application layer when the transaction has completed or an error occurs.

Inputs: psReqPkt – Pointer to the request structure containing the information needed to create and complete a 1394 async transaction.

Outputs: RtnVal – SBI_NO_ERROR (zero) signifies successful completion of this function. A non-zero return indicates error condition. See error code list in Appendix A.

Prototype:

USHORT sbiAsyncWrite(PASYNC_RWPKT psReqPkt);

6.7.3 sbiGetElapsedCycleTime

Description: This function provides the application the capability of determining how long an action has taken using the API firmware. The cycle timer of the TSB43AA82 is used as the timer source for

evaluating the elapsed time. Given some initial cycle time value, this function will read the current cycle time value to determine the elapsed time returned in the same format as the cycle time register. As an example, given a returned value of 0x02416A7D would imply (1 * 1 sec) + (0x416 * 125us) + (0xA7D * 40ns) or 1 second plus 130 milliseconds plus 857 microseconds plus 400 nanoseconds of elasped time

Inputs: qStartCycleTime – The initial cycle time value to compute the elapsed time from.

Outputs: RtnVal – the elapsed time between the input cycle time and the current cycle time.

Prototype:

QUADLET sbiGetElapsedCycleTime(QUADLET qStartCycleTime);

6.7.4 sbiCauseBusReset

Description: This function is provided by the API to allow the application to generate a reset on the bus if there is a need to reset the system in order to become functional again.

Inputs: None

Outputs: None

Prototype:

void sbiCauseBusReset(void);

6.7.5 sbiGetSelfIDs

Description: This function is provided by the API to allow the application to get to the list of self-ids that were generated on the last bus reset. The end of the self-ids is indicated by a quadlet containing 0xAAAAAAAA.

Inputs: None

Outputs: RtnVal – A pointer to the API maintained list of self-ids.

Prototype:

PULONG sbiGetSelfIDs(void);

6.7.6 sbiCrcCalculate

Description: This function is provided by the API firmware to give the application the capability to verify the config rom of a remote node.

Inputs: buf – A pointer to the buffer containing the quadlet aligned CRC data.
nNumQuads – The number of quadlets of data to compute the CRC for.
CRC1991 – True if the CRC is to be computed by the 1991 version.

Outputs: RtnVal – The 16 bit CRC value for the data provided.

Prototype:

```
USHORT sbiCrcCalculate( PULONG buf, int nNumQuads, BOOL CRC1991 );
```

6.8 DMA Prototypes

6.8.1 SphnxGetNumQuadsInDxF

Description: This function is provided by the API and would be used by the application when processing data through the DATA FIFOs. It allows the application firmware to determine how much data can be pulled out of the Data Receive FIFO or how much data can be put into the Data Transmit FIFO. The **#defines** are used to help simplify and clarify what action is taking place.

Inputs: inDTF – True if wanting to check how much space is available in the DTF.

Outputs: RtnVal – The number of quadlets that could be read or stored into the FIFO.

Prototype:

```
#define SphnxGetNumQuadsInDRF()  SphnxGetNumQuadsInDxF(0)
#define SphnxGetNumQuadsInDTF()  SphnxGetNumQuadsInDxF(1)

USHORT SphnxGetNumQuadsInDxF( BOOL inDTF );
```

6.8.2 SphnxAPIDMARWSetup

Description: The API provides this function to allow the application to setup the hardware DMA engine to transfer data but does not initiate the transfer. Transfers are based on 1394 transactions, a write operation implies data flows from this node to a remote node, a read operation will be from the remote node to this node. Based on the direction of data flow, the FIFO sizes will be adjusted to aid in the movement of the data. There are several modes of operation that allow the application some flexibility in transferring the data. In HOST mode, the application would manually setup a request, the FIFO would receive the data and it would be manually pulled or the entire packet would loaded for

transfer to a remote node. In "PIO" modes, the DMA hardware is used to create 1394 requests and data is manually transferred to/from the FIFOs. In "BDI" modes, the DMA hardware is used again and data is piped through the Bulky Data Interface. The **#defines** are used to help simplify and clarify what action is taking place.

Inputs: RDop – True if the DMA is being setup for a 1394 READ operation that will receive data from another node, False if a 1394 WRITE operation that will transmit data to another node.
pReq – Pointer to the structure containing the information needed to setup the DMA as well as the mode of operation.

Outputs: Status – Returned through the DMA structure, SBI_NO_ERROR (zero) signifies successful completion of this function. A non-zero return indicates error condition. See error code list in Appendix A.

Prototype:

```
#define SphnxAPIDMARdSetup(pReq)  SphnxAPIDMARWSetup( 1, pReq )
#define SphnxAPIDMAWrSetup(pReq)  SphnxAPIDMARWSetup( 0, pReq )

void SphnxAPIDMARWSetup( BOOL RDop, PDMA_REQUEST pReq );
```

6.8.3 SphnxAPIDMARWExec

Description: The API provides this function to allow the application to initiate a hardware DMA transaction that was setup up by SphnxAPIDMARWSetup or to continue a DMA operation that halted due to a timeout or error condition. The **#defines** are used to help simplify and clarify what action is taking place.

Inputs: RDop – True if the DMA was setup for a 1394 READ operation that will receive data from another node, False if a 1394 WRITE operation that will transmit data to another node.
InitStart – True if the DMA operation is being initiated for the first time, False if the operation is being restarted after being halted for some reason.

Outputs: None

Prototype:

```
#define SphnxAPIDMAWrCont()  SphnxAPIDMARWExec( 0, 0 )
#define SphnxAPIDMAWrExec()  SphnxAPIDMARWExec( 0, 1 )
#define SphnxAPIDMARdCont()  SphnxAPIDMARWExec( 1, 0 )
#define SphnxAPIDMARdExec()  SphnxAPIDMARWExec( 1, 1 )
```

void SphnxAPIDMARWExec(BOOL RDop, BOOL InitStart);

6.8.4 SphnxAPIDMARWRqst

Description: The API provides this function to allow the application to set up and initiate a hardware DMA transaction. This function combines the setup and executes operations in one call. The **#defines** are used to help simplify and clarify what action is taking place.

Inputs: RDop – True if the DMA is being setup for a 1394 READ operation that will receive data from another node, False if a 1394 WRITE operation that will transmit data to another node.
pReq – Pointer to the structure containing the information needed to setup the DMA as well as the mode of operation.

Outputs: Status – Returned through the DMA structure, SBI_NO_ERROR (zero) signifies successful completion of this function. A non-zero return indicates error condition. See error code list in Appendix A.

Prototype:

```
#define SphnxAPIDMARead(pReq)   SphnxAPIDMARWRqst( 1, pReq )
#define SphnxAPIDMAWrite(pReq)  SphnxAPIDMARWRqst( 0, pReq )

void SphnxAPIDMARWRqst( BOOL RDop, PDMA_REQUEST pReq );
```

6.8.5 SphnxAPIAbortDma

Description: This function is provided by the API to allow the application to abort and clean up an ongoing DMA operation. The **#defines** are used to help simplify and clarify what action is taking place.

Inputs: RDop – True if the DMA is or was doing a 1394 READ, False if a 1394 WRITE operation.

Outputs: None

Prototype:

```
#define SphnxAPIAbortDmaWrite() SphnxAPIAbortDma(0)
#define SphnxAPIAbortDmaRead()  SphnxAPIAbortDma(1)

void SphnxAPIAbortDma( BOOL RDop );
```

6.8.6 SphnxAPIGetPktizrStatus

Description: This function is provided by the API to allow the application to retrieve the packetizer status after a DMA operation has completed. This provides the application a way to determine why the DMA operation finished and whether it can be restarted. When describing the DMA_CB_DATA structure, there are **#defines** on the packetizer status. These defines are used to help simplify parsing of the status which is the 1st quadlet of the four.

Inputs: pul4Quads – A pointer to storage for 4 quadlets of data as defined in the data manual (registers 0xE8-0xF4).

Outputs: None

Prototype:

void SphnxAPIGetPktizrStatus(PULONG pul4Quads);

6.8.7 SphnxAPIPIOGetPacket

Description: This function is provided by the API to allow the application to retrieve an entire packet contained in the DRF. This could be used when the DMA mode was HOST or "PIO" and the data is a complete packet.

Inputs: pulDataBuf – pointer to a buffer to store the data. The application should provide a buffer large enough for header and data.

Outputs: None

Prototype:

void SphnxAPIPIOGetPacket(PULONG pulDataBuf);

6.8.8 SphnxAPIPIOGetPcktHedr

Description: This function is provided by the API to allow the application to retrieve only the header data for an entire packet contained in the DRF. This could be used when the DMA mode was HOST or "PIO" and the data is a complete packet.

Inputs: pul5Quads – pointer to a buffer to store the data for the header. The header is as defined in the data manual, 1 quadlet of status information followed by the 4 quadlets of the standard 1394 packet header.

Outputs: None

Prototype:

void SphnxAPIPIOGetPcktHedr(PULONG pul5Quads);

6.8.9 SphnxAPIPIOGetPcktData

Description: This function is provided by the API to allow the application to retrieve the data for a packet contained in the DRF. This could be used after the packetizer pulls in a packet and strips the header or if the header data had been read already.

Inputs: NumQuads – The number of quadlets to read from the DRF.
pulDataBuf – pointer to a buffer to store the data. The application should should provide a buffer large enough for the data as indicated by NumQuads.

Outputs: None

Prototype:

void SphnxAPIPIOGetPcktData(USHORT NumQuads, PULONG pulDataBuf);

6.8.10 SphnxAPIPIOGetDataQuad

Description: This function is provided by the API to allow the application to read data from the DRF a single quadlet at a time. It should be noted that the number of quadlets contained in the FIFO should be checked before each data read and no read should be performed if there are zero quadlets available.

Inputs: None

Outputs: RtnVal – The current quadlet available from the host interface to the DRF.

Prototype:

ULONG SphnxAPIPIOGetDataQuad(void);

6.8.11 SphnxAPIPIOUnPause

Description: This function is provided by the API to allow the application to notify the packetizer to request the next packet while removing the current packet from the DRF. This function is only valid if "USE_AUTOPAUSE" is defined. This allows the application to regulate the flow of data being received.

Inputs: None

Outputs: None

Prototype:

void SphnxAPIPIOUnPause();

6.8.12 SphnxAPIPIOWriteCont

Description: This function is provided to allow the application to manually load packet data into the DTF. This would be used in conjunction with SphnxAPIPIOWriteUpdt. The packet header and all but the last quadlet of data would be loaded using this function. It is up to the application to ensure there is enough space in the FIFO to load the data.

Inputs: Quad – A quadlet of data to be stored in the DTF FIFO.

Outputs: None

Prototype:

void SphnxAPIPIOWriteCont(ULONG Quad);

6.8.13 SphnxAPIPIOWriteUpdt

Description: This function is provided to allow the application to complete manual loading of packet data into the DTF. This would be used after SphnxAPIPIOWriteCont. The last quadlet of the packet header or data would be loaded using this function. It is up to the application to ensure there is enough space in the FIFO to load the data.

Inputs: Quad – The final quadlet of data to be stored in the DTF FIFO for a packet.

Outputs: None

Prototype:

void SphnxAPIPIOWriteUpdt(ULONG Quad);

6.9 SBP-2 Prototypes

6.9.1 SphnxAPIGetCmdAgent

Description: This function is provided by the API to allow the application to determine the Agent ID associated with a Command ORB.

Inputs: pCmdOrb – Pointer to the Command ORB to determine the Agent ID number.

Outputs: RtnVal – The Agent ID associated with the Command ORB or –1 if no ID available.

Prototype:

USHORT SphnxAPIGetCmdAgent(PSBP2CMND_ORB pCmdOrb);

6.9.2 SphnxAPIGetCmdGenCount

Description: This function is provided by the API to allow the application to use the generation count of a Command ORB with various other transactions that may be needed to process it.

Inputs: pCmdOrb – Pointer to the Command ORB to determine the generation count.

Outputs: RtnVal – The generation count associated with the Command ORB.

Prototype:

ULONG SphnxAPIGetCmdGenCount(PSBP2CMND_ORB pCmdOrb);

6.9.3 SphnxAPISetCmdDepStatus

Description: This function is provided by the API to allow the application to update the Command Dependant Status (CDS) block for a command agent. This additional status information will be sent when the SBP-2 status is sent back to the initiator node.

Inputs: usCmdAgt – Agent ID to have its CDS block updated.
usByteCnt – The length in bytes of the CDS block. The maximum is 24 for a total of six additional quadlets.
PCmdDepStatus – Pointer to the array of quadlets containing the CDS.

Outputs: RtnVal – SBI_NO_ERROR (zero) signifies successful completion of this function. A non-zero return indicates error condition. See error code list in Appendix A.

Prototype:

```
void SphnxAPISetCmdDepStatus(   USHORT usCmdAgt,
                                USHORT usByteCnt,
                                PULONG pCmdDepStatus );
```

6.9.4 SphnxAPICmdTerminate

Description: This function is provided by the API to allow the application to send the SBP-2 status block and any Command Dependant Status back to the initiator node. This function also resets the hardware to fetch the next command ORB for the command agent.

Inputs: uStatus – Desired SBP-2 status as defined in the spec. The agent will transition to the dead state if the "d" bit is set, the "src" and "len" fields will be updated by the API.
pCmdOrb – Pointer to the Command ORB that has completed processing.

Outputs: None

Prototype:

```
void SphnxAPICmdTerminate( USHORT uStatus, PSBP2CMND_ORB pCmdOrb );
```

6.9.5 SphnxAPISendUnsolicitedStatus

Description: This function is provided by the API to allow the application to send an unsolicited status to the initiator node if it has been enabled.

Inputs: uStatus – Desired SBP-2 status as defined in the spec. The agent will transition to the dead state if the "d" bit is set, the "src" and "len" fields will be updated by the API.
pCmdOrb – Pointer to the Command ORB that has completed processing.

Outputs: None

Prototype:

```
void SphnxAPISendUnsolicitedStatus( USHORT uStatus, PSBP2CMND_ORB pCmdOrb );
```

6.10 Control and Status Registers

The following are the CSR registers supported by LynxSoft API:

```
/* Core CSR offsets. */
#define STATE_CLEAR                          0x0000
#define STATE_SET                            0x0004
#define NODE_IDS                             0x0008
#define RESET_START                          0x000C
#define SPLIT_TIMEOUT_HI                     0x0018
#define SPLIT_TIMEOUT_LO                     0x001C

/* Serial bus dependent offsets. */
#define CYCLE_TIME                           0x0200
#define BUS_TIME                             0x0204
#define BUSY_TIMEOUT                         0x0210
#define BUS_MANAGER_ID                       0x021C
#define BANDWIDTH_AVAILABLE_REGISTER         0x0220
#define CHANNELS_AVAILABLE_HI                0x0224
#define CHANNELS_AVAILABLE_LO                0x0228
#define MAINT_UTILITY                        0x0230
```

7. Hardware Abstraction Layer (HAL)

7.1 Introduction

The Hardware Abstraction Layer (HAL) is the lowest layer in the LynxSoft architecture. It provides an interface to the link layer hardware to abstract the upper layer firmware modules from the specifics of the hardware. This layer provides no protocol intelligence (1394 or SBP-2), minimal event notification and provides an interface to use the hardware acceleration features for SBP-2.

This section of the document provides an overview of the interface (defines, callbacks, structures, and prototypes) that the HAL provides to the upper layers. The architecture of the expected use for the iSphynxII LynxSoft HAL is shown in the diagram below.

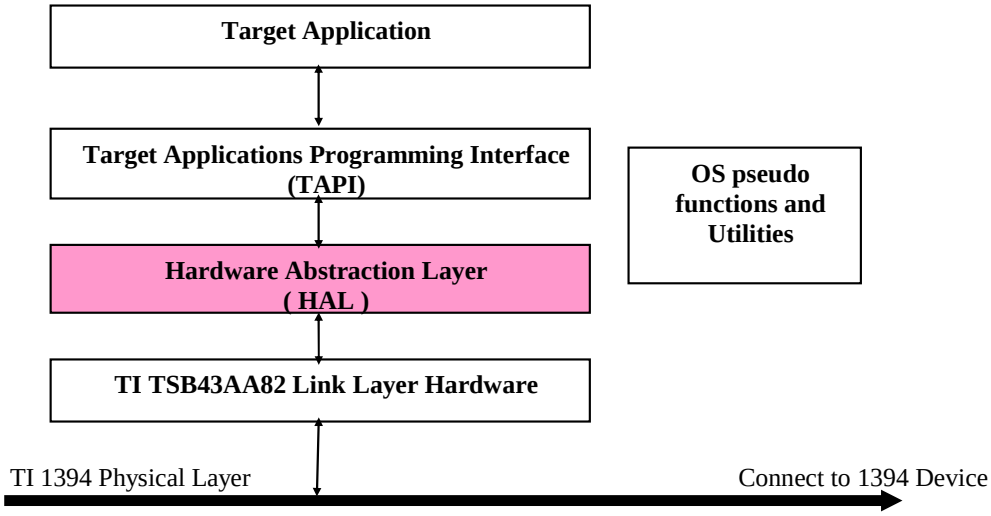


Figure 6 LynxSoft Target Firmware Architecture (HAL view)

Deleted: <sp><sp><sp><sp><sp><sp><sp><sp><sp>

7.2 Defines

The following chart describes the compiler defines used in the TSB43AA82 HAL interface. Note that this does *not* describe all defines used in the HAL, only ones that are relevant to the upper layers.

Define	Description
MAX_COMMAND_AGENT	The maximum number of command agents supported. This value should be between 0 and 4.
COMMAND_BLOCK_SIZE_QUADS	Size of command blocks in units of quadlets.
AUTO_CONFIG_ROM	If defined, Config Rom data will be loaded into HW and managed by it
COMMAND_TX_QUE_SIZE	Size of CTQ in units of quadlets.
COMMAND_RX_FIFO_SIZE	Size of CRF in units of quadlets.
ASYNC_TX_FIFO_SIZE	Size of ATF in units of quadlets.
ASYNC_RX_FIFO_SIZE	Size of ARF in units of quadlets.
DMA_DXF_TOTAL_QUADS	Total FIFO space for DTF/DRF in units of 4 quadlets
DMA_DXF_MAX_SIZE	Max size either DTF or DRF will be set to when it is being used
DMA_DXF_MIN_SIZE	Left over FIFO space either DTF or DRF will be set to when not in use
DMA_WRITE_FIFO_SIZE	Initial Size of DTF in units of 4 quadlets.
DMA_READ_FIFO_SIZE	Initial Size of DRF in units of 4 quadlets.
DMA_MODE_HOST	Use HOST access to FIFO data
DMA_MODE_BDIDPP	Use BDIF access to FIFO using DPP
DMA_MODE_PIODPP	Use HOST access to FIFO using DPP
DMA_MODE_BDIPKT	Use BDIF access to FIFO using Packetizer
DMA_MODE_PIOPKT	Use HOST access to FIFO using Packetizer
DMA_MODE_UNK	Validity Check
EnableInterrupts	Macro to enable Maskable Interrupts on Host
DisableInterrupts	Macro to disable Maskable Interrupts
MASKABLE_INT_ENABLED	Macro to check if MI is enabled
CtMillisec	Macro to provide Millisecond delay function
LYNXHAL_NO_ERRORS	No errors to report.
LYNXHAL_ERROR_SELFID	No data in FIFO when trying to return self - ids.
LYNXHAL_ERROR_ATF_NOT_EMPTY	ATF not empty when sending Phy packet.
LYNXHAL_FIFO_EMPTY	No data to return from FIFO.
LYNXHAL_FIFO_NOT_EMPTY	After host removes a packet from the FIFO, the FIFO still contains data.
LYNXHAL_ERROR_DMA_BUSY	Data FIFO is in busy state.

LYNXHAL_ERROR_DMA_PEND	Data FIFO is in pending state.
LYNXHAL_ERROR_DMA_PAGE_FAULT	Data FIFO is in page fault state.
LYNXHAL_ERROR_TRANSACTION	Transaction completed with an error.
LYNXHAL_ERROR_PACKET_VALUE	Packet to large for FIFO
LYNXHAL_ERROR_BUSRESET	BusReset occurred during operation

7.3 Structures

This section describes the data structures used by the HAL layer to provide information to the upper layer or for the upper layer to provide information to the HAL.

7.3.1 Busreset Info Structure

The following is the format of the Busreset Info Structure:

```
typedef struct {
    ULONG   ulBusResetCount;
    ULONG   ulResetCycleTime;
    USHORT  usBusResetError;
    UINT    uiNodeID;
    UINT    uiBusID;
    ULONG   ulNodeCount;
    USHORT  bRoot;
 } BUSRESETINFO, *PBUSRESETINFO;
```

- **ulBusResetCount** is a counter of bus resets and is also used as the current generation value.
- **ulResetCycleTime** is the cycle timer value of the last reset. This value is taken during the interrupt service routine. The recorded time is slightly after the actual event.
- **ucBusResetError** is the 4-bit bus reset error code provided by the link.
- **uiNodeID** is the current 6-bit node id.
- **uiBusID** is the current 10-bit bus id.
- **ulNodeCount** is the current number of nodes active on the bus.
- **bRoot** is TRUE if this node is root.

7.3.2 Config Rom Init Structure

Information needed to initialize CONFIG ROM in hardware.

```
typedef struct _ConfigRomInit {
  ULONG   ulMgtAgentAddr;
  QUADLET qSerialNumber;
  PULONG  pulCFRData;
} CFGROMINIT, *PCFGROMINIT;
```

- **ulMgtAgentAddr** is the 1394 Offset Low for the Management Agent Register.
- **qSerialNumber** is the Serial # to add to Config Rom.
- **pulCFRData** is the pointer to the storage containing CFR data that needs to be updated with the serial # and the CRC values for each block.

7.3.3 HAL Info Structure

The following is the format of the HAL Info Structure:

```
typedef struct {
    // Inputs from caller
    PFN     pBusResetCallback;
    PFN     pIndicationCallback;
    PFN     pMngtORBCallback;
    PFN     pCmdORBCallback[4];

    // Outputs to caller
    LYNXHAL_MNGT_ORB   ManagementORB;
    LYNXHAL_CMD_ORB     CommandORB[4];
    USHORT        bSelfIDEnd;
    PULONG        pulSelfIDs;
    BUSRESETINFO  BRInfo;
} LYNXHAL_INFO;
```

The first four entries in this structure are pointers to the callbacks mentioned in section 7.4.

- **ManagementORB** is a structure that is filled in by the HAL after receiving an ORB in the MRF and before calling pMngtORBCallback.
- **CommandORB** is an array of 4 command ORB structures (one for each command agent). This is filled in by the HAL after receiving an ORB in the CRF and before calling the corresponding pCmdORBCallback.
- **bSelfIDEnd** is set when the self id period ends. At this point the upper layers may call the function to store the self id packets.
- **PulSelfIDs** is a pointer to the storage location provided to store the self-ids. 0xAAAAAAAA denotes end of self-id quadlets
- **BRInfo** is a structure as described above.

7.3.4 Management ORB Structure

The following is the format of the management ORB structure:

```
typedef struct
{
     UCHAR     ucStatus;
     UCHAR     ucrCode;
     UCHAR     ucSpeed;
     UCHAR     ucTlabel;
     UCHAR     ucAck;
     USHORT    usSourceID;
     USHORT    usOffsetHi;
     ULONG     ulOffsetLow;
     ULONG     ulGenerationCnt;
     ULONG     ulRxTime;
     ULONG     *ulManagementORB;
} LYNXHAL_MNGT_ORB;
```

- **ucStatus** is the status field in the packet trailer.
- **ucrCode** is the response code for this packet.
- **ucSpeed** is the speed field for this packet.
- **ucTlabel** is the transaction label for this packet.
- **ucAck** is the returned acknowledge.
- **usSourceID** is the node ID of the initiator.
- **usOffsetHi** is the ORB offset high of the fetched ORB.
- **ulOffsetLow** is the ORB offset low of the fetched ORB.
- **UlGenerationCnt** is the Generation count at time the ORB was received.
- **UlRxTime** is the cycle time at the time the HAL layer received the ORB.
- **ulManagementORB** is the ULONG array where the 32 byte ORB data will be stored.

7.3.5 Command ORB Structure

The following is the format of the command ORB structure:

```
typedef struct
{
  UCHAR   ucStatus;
  UCHAR   ucrCode;
  UCHAR   ucSpeed;
  UCHAR   ucTlabel;
  UCHAR   ucNextTlabel;
  UCHAR   ucAck;
  USHORT  usSourceID;
  USHORT  usOffsetHi;
  ULONG   ulOffsetLow;
  ULONG   *ulCommandORB;
} LYNXHAL_CMD_ORB;
```

These fields have the same meaning as the LYNXHAL_MNGT_ORB.

7.3.6 Async Packet Transmit Structure

The following is the format of the async packet transmit structure:

```
typedef struct
{
    UCHAR      ucAckReturned;
    UCHAR      ucSpeed;
    UCHAR      ucTlabel;
    UCHAR      ucRetryCode;
    UCHAR      ucTCode;
    UCHAR      ucPriority;
    USHORT     usDestID;
```

```
    UCHAR      ucRCode;
    USHORT     usOffsetHi;
    ULONG      ulOffsetLo;
    USHORT     usDataLength;
    USHORT     usExtTCode;
    ULONG ulGenerationCnt;
    ULONG      *ulData;
} LYNXHAL_ASYNC_PACKET_TX;
```

- **ucAckReturned** is the acknowledge packet returned for this packet.
- **ucSpeed** is the 2-bit speed field for the packet.
- **ucTlabel** is the 6-bit transaction label for the packet.
- **ucRetryCode** is the 2-bit retry code for the packet.
- **ucTCode** is the 4-bit transaction code.
- **ucPriority** is the 4-bit priority field (should be zero).
- **UsDestID** is the 16-bit destination ID for the packet.
- **ucRCode** is the response code if this is a response packet, 0xFF if not a response.
- **usOffsetHi** is the 16-bit address offset high of the packet (if request).
- **ulOffsetLow** is the 32-bit address offset low of the packet (if request).
- **usDataLength** is the 16-bit data length of the packet.
- **usExtTCode** is the 16-bit extended transaction code of the packet.
- **ulGenerationCnt** the generation count for the packet to prevent stale packets on bus.
- **ulData** is the ULONG array of data to send with this packet.

7.3.7 Async Packet Receive Structure

The following is the format of the async packet receive structure:

```
typedef struct
{
    UCHAR      ucStatus;
    UCHAR      ucSpeed;
    UCHAR      ucAck;
    USHORT     usDestID;
    UCHAR      ucTlabel;
    UCHAR      ucRetryCode;
    UCHAR      ucTCode;
    UCHAR      ucPriority;
    USHORT     usSourceID;
    UCHAR      ucRCode;
    USHORT     usOffsetHi;
    ULONG      ulOffsetLo;
    USHORT     usDataLength;
    USHORT     usExtTCode;
    ULONG ulGenerationCnt;
```

```
    ULONG ulCycleTime;
    ULONG      *ulData;
} LYNXHAL_ASYNC_PACKET_RX;
```

- **ucStatus** is the status field in the packet trailer.
- **ucSpeed** is the speed field for this packet.

- **ucAck** is the returned acknowledge.
- **usDestID** is the node ID of the destination.
- **ucTlabel** is the transaction label for this packet.
- **ucRetryCode** is the 2-bit retry code for the packet.
- **ucTCode** is the 4-bit transaction code.
- **ucPriority** is the 4-bit priority field (should be zero).
- **usSourceID** is the node ID of the sender.
- **ucRCode** is the response code for this packet (if response).
- **usOffsetHi** is the address offset high of the packet (if request).
- **ulOffsetLow** is the address offset low of the packet (if request).
- **usDataLength** is the 16-bit data length of the packet.
- **usExtTCode** is the 16-bit extended transaction code of the packet.
- **ulGenerationCnt** the generation count for the packet to prevent stale packets on bus.
- **ulCycleTime** the CycleTimer value at the time a received packet was read from the FIFo.
- **ulData** is the ULONG array of data to send with this packet.

7.3.8 Data FIFO Control Structure

The following is the format of the DTF/DRF control structure:

```
typedef struct
{
    USHORT          bNotify;
    USHORT          usSpeed;
    USHORT          ucMaxPayload;
    USHORT          bPageEnable;
    USHORT          ucPageSize;
    USHORT          usBlockSize;
    ULONG           ulBlockCount;
    USHORT          usDestID;
    USHORT          usOffsetHi;
    ULONG           ulOffsetLo;
    USHORT          usPageCount;
    ULONG      ulGenerationCnt;
    PFNDXFCB        pPcktEndCallback;
    PFNDXFCB        pDataEndCallback;
} LYNXHAL_DTF_DRF_CONTROL;
```

- **bNotify**, when TRUE tells the hardware to automatically send transaction status.
- **ucSpeed** is the speed field for the data packets.
- **ucMaxPayload** is the 4-bit max payload, max data length = 2^(ucMaxPayload+2) bytes.
- **bPageEnable** enables page fetching.
- **ucPageSize** is the page size filed from the ORB.
- **usBlockSize** is the 16-bit block size of data in bytes.
- **ulBlockCount** is the 30-bit block count of data.

- **usDestID** is the 16-bit destination ID of the initiator.
- **usOffsetHi** and **usOffsetLo** is the 48-bit address of the data descriptor.
- **usPageCount** is the total number of pages, if any.
- **ulGenerationCnt** the generation count for the packet to prevent stale packets on bus.
- **pPcktEndCallback** is the callback to call when the DMA transfer completes 1 block transfer in DMA_MODE_PIOPKT.
- **pDataEndCallback** is the callback to call when the DMA transfer is complete.

7.4 Callbacks

There are a total of five types of callbacks supported in the TSB43AA82 HAL. These callbacks are provided by the upper layers using the structures defines in section 3. If the upper layers do not wish to implement a certain callback, a NULL pointer can be passed into the HAL and the callback will not be executed. The following sections describe these callbacks in detail.

7.4.1 Bus Reset Callback

This callback is executed when a bus reset occurs. The bus information in the HAL info structure defined in the previous section is updated at this time. This function will only be called after the Self-IDs have been received and stored. It is possible on a multi-busreset that you would have more than one call in the queue but an attempt is made to minimize the number of calls placed on to the queue. The upper layer would need to perform any processing that is needed after a busreset has occurred.

7.4.2 Indication Callback

This callback is executed when a packet is received in the Asynchronous Receive FIFO (ARF). The upper layer is expected to remove the packet from the FIFO through the use of two functions: LynxHALSaveAsyncHeader and LynxHALSaveAsyncData. The upper layer must evaluate the packet and determine what to do with it. A valid response packet is required to complete the Async request.

7.4.3 Management ORB Callback

This callback is executed when the Management ORB End interrupt is received. The management ORB structure is updated at this time. The upper layer needs to determine what management action is requested and perform the processing. The upper layer must complete the management ORB request by issuing a call to the HAL function LynxHALClearMgmtAgentBusy. This will reset the hardware to allow another management ORB request be made.

7.4.4 Command ORB Callback (4)

Four separate command ORB callbacks are supported or one as the callback takes a parameter which is the agent number. Each command agent supports its own callback. This callback is executed when the Command ORB End interrupt is received. The command ORB structure is updated at this time. The upper layer needs to determine what command action is requested and perform the processing. The upper layer must complete the command ORB request by issuing a call to the HAL function LynxHALSetAgentReady. This will reset the hardware to allow another command ORB request be made to this command agent.

7.4.5 Data End Callback

This is called when a DMA transaction is complete. This callback takes in a parameter, an unsigned short, that is a status composed of 3 items. The DxF Acknowledge register (0xA0) status for the DTF or DRF, the DxF Packetizer Control bits (0,1), and the Transaction Error bit from the Transaction Timer Control at 0x60. The following defines are used to extract the specific information desired and a prototype is as defined for DMA callbacks (ie DTF/DRF).

```
#define DxF_AckReg_Mask     0x81F1  /* 16 bits (either DTF or DRF) */
#define DxF_AckValid(DxFStat) ((DxFStat&0x0001)   )
#define DxF_DmaLRsv(DxFStat) ((DxFStat&0x000E)>>1)
#define DxF_AckRecvd(DxFStat) ((DxFStat&0x00F0)>>4)
#define DxF_AckError(DxFStat) ((DxFStat&0x0100)>>8)
#define DxF_PktCntrl(DxFStat) ((DxFStat&0x0600)>>9)
#define DxF_DmaMRsv(DxFStat) ((DxFStat&0x7800)>>11)
#define DxF_DmaError(DxFStat) ((DxFStat&0x8000)>>15)

void DataEndCallback( USHORT usStat  );
```

7.5 Basic Prototypes

The following sub-sections describe the basic interface functions provided in the TSB43AA82 HAL.

7.5.1 LynxHALGetVersion

Description: This function returns the HAL version number in binary coded decimal (rrmmddyy).

Inputs: None

Outputs: pqSoftwareVersion – The HAL's software version information (rrmmddyy) rev, month, day, yy in binary coded decimal.
pqHardwareVersion – The hardware's version information (vvvvvvrr), where v is the Vendor ID and r is the hardware revision.

Prototype:

```
void LynxHALGetVersion(   QUADLET* pqSoftwareVersion,
                          QUADLET* pqHardwareVersion );
```

7.5.2 LynxHALInit

Description: This function initializes the link core and HAL layer. This function must be called prior to any other HAL function.

Inputs: *pHALInfoStruct - Pointer to HAL info structure
pCnfgInfo – Pointer to structure containing the information needed to initialize config rom.

Outputs: None

Prototype:

```
void LynxHALInit( LYNXHAL_INFO *pHALInfoStruct; ,
                                    PCFGROMINIT pCnfgInfo );
```

7.5.3 LynxHALTerminate

Description: This function disables 1394 interrupts and turns the management/command agents of and disables the maskable interrupt to the microcontroller.

Inputs: None

Outputs: None

Prototype:

void LynxHALTerminate( );

7.5.4 LynxHALGetCycleTime

Description: This function sends a phy config or link on packet.

Inputs: None

Outputs: RtnVal - 32-bit cycle timer value.

Prototype:

ULONG LynxHALGetCycleTime(void);

7.5.5 ctDelay

Description: This function provides a software delay based upon the cycle timer of the TSB43AA82 HW. The delay is in Iso cycles (ie every 125 micro seconds).

Inputs: N_x_125usecs – the number of Iso cycles of delay.

Outputs: None

Prototype:

void ctDelay(ULONG N_x_125usecs);

7.5.6 ctMillisec

Description: This function provides a software delay based upon the cycle timer of the TSB43AA82 HW. The delay is every 8 Iso cycles (ie every 1 milli-seconds).

Inputs: Millisecs – the number of milliseconds of delay.

Outputs: None

Prototype:

```
#define CyclesPerMsec  8

#define ctMillisec(  Millisecs )   ctDelay( (ULONG)CyclesPerMsec * Millisecs )
```

7.5.7 ctMicrosec

Description: This function provides a software delay based upon the cycle timer of the TSB43AA82 HW. The delay is every 1 micro-second.

Inputs: Microsecs – the number of microseconds of delay.

Outputs: None

Prototype:

```
void ctMicrosec(  ULONG Microsecs );
```

7.6 1394 Prototypes

The following sub-sections describe the 1394 interface functions provided in the TSB43AA82 HAL.

7.6.1 LynxHALSendAsyncPacket

Description: This function sends an asynchronous packet through the ATF.

Inputs: *psPacket - Pointer to async packet structure

Outputs: RtnVal – 0 indicates transaction successful, non-zero indicates a transaction error.

Prototype:

```
USHORT LynxHALSendAsyncPacket( LYNXHAL_ASYNC_PACKET_TX *psPacket );
```

7.6.2 LynxHALSendPhyPacket

Description: This function sends a phy config or link on packet.

Inputs: ulPhyPacket - Phy packet data

Outputs: 0 - packet has been loaded into ATF
Other – error

Prototype:

int LynxHALSendPhyPacket(ULONG ulPhyPacket);

7.6.3 LynxHALSaveAsyncHeader

Description: This function pulls a packet header out of the ARF and places it in memory. If the packet has data, LynxHALSaveAsyncData should be called after this function. This allows dynamic memory allocation.

Inputs: *psPacket - Storage for packet RX structure

Outputs: 0 - Successful and FIFO empty
Other - error or FIFO not empty

Prototype:

int LynxHALSaveAsyncHeader(LYNXHAL_ASYNC_PACKET_RX *psPacket);

7.6.4 LynxHALSaveAsyncData

Description: This function pulls async data out of the ARF and places it in memory. This function is called immediately after LynxHALSaveAsyncData. This allows dynamic memory allocation.

Inputs: *psPacket - Storage for packet RX structure

Outputs: 0 - Successful and FIFO empty
Other - error or FIFO not empty

Prototype:

int LynxHALSaveAsyncData(LYNXHAL_ASYNC_PACKET_RX *psPacket);

7.6.5 LynxHALCauseBusReset

Description: This function pulls a packet out of the ARF and places it in memory.

Inputs: iGapCount - Gap count to set before reset. The call to this function after sending a Phy Config packet must have iGapCount set to the new Gap Count value in the Phy Config packet. All other calls to this function must have iGapCount to 0x3F, the maximum Gap Count value.

Outputs: None

Prototype:

void LynxHALCauseBusReset(UINT iGapCount);

7.6.6 LynxHALGetThisNodesID

Description: This function returns the bus ID/node ID.

Inputs: None

Outputs: RtnVal - 16-bit Bus ID and Node ID value.

Prototype:

USHORT LynxHALGetThisNodesID(void);

7.6.7 LynxHALGetBusID

Description: This function returns the bus ID from the iSphynx Bus Reset register.

Inputs: None

Outputs: RtnVal - Bus ID in same format as IEEE 1394 NODE_IDS register.

Prototype:

ULONG LynxHALGetBusID(void);

7.6.8 LynxHALSetBusID

Description: This function sets the bus ID in the iSphynx Bus Reset register.

Inputs: ulNewBusID - Bus ID in same format as IEEE 1394 NODE_IDS register.

Outputs: None

Prototype:

void LynxHALSetBusID(ULONG ulNewBusID);

7.6.9 LynxHALGetSplitTimeout

Description: This function returns the current split timeout value set in the link.

Inputs: None

Outputs: RtnVal -16-bit Split Timeout value in 125us units

Prototype:

USHORT LynxHALGetSplitTimeout(void);

7.6.10 LynxHALSetSplitTimeout

Description: This function sets the current split timeout value in the link.

Inputs: usValue - 16-bit value to set in units of 125us

Outputs: None

Prototype:

void LynxHALSetSplitTimeout(USHORT usValue);

7.7 DMA Prototypes

The following sub-sections describe the DMA interface functions provided in the TSB43AA82 HAL.

7.7.1 LynxHALBDISetup

Description: This function setups the BDI for the MODE of transfer to use (ie Device Signals to setup, allowed modes). This function is normally only called during initialization.

Inputs: SigEnable – Allowable signals to be enabled
ucBulkyMode - BDIF modes 0 (A) to 7 (H)

Outputs: None

Prototype:

void LynxHALBDISetup(ULONG SigEnable, UCHAR ucBulkyMode);

7.7.2 LynxHALDMASetup

Description: This function setups the DMA for the MODE of transfer to use (ie Host, DPP, Packetizer) as well as the direction of data (transmit or receive)

Inputs: bDMARead - True if Receiving data through DRF interface
ucDMAMode - Define DMA mode of operation (DMA_MODE_xxxx).

Outputs: None

Prototype:

void LynxHALDMASetup(BOOL bDMARead, UCHAR ucDMAMode);

7.7.3 LynxHALHostDMAWrite

Description: This function loads the DTF using the host interface. The data is expected to contain the header quadlets in the required format as specified by the iSphynx data manual. LynxHALDMASetup() should be called to setup hardware for host access to DTF before transmitting data.

Inputs: *pulData - Pointer to array of data
uiLengthInQuads - Length of packet in quads

Outputs: None

Prototype:

void LynxHALHostDMAWrite(ULONG *pulData, UINT uiLengthInQuads);

7.7.4 LynxHALHostDMARead

Description: This function removes one packet from the DRF and places in memory.

LynxHALDMASetup() should be called to setup hardware for host access to DRF before receiving data.

Inputs: *ulData - Storage for data read from DRF (one packet max is 517 quads)

Outputs: RtnVal - Amount of data read in quadlets

Prototype:

USHORT LynxHALHostDMARead(ULONG *ulData);

7.7.5 LynxHALCheckDMAState

Description: This function checks the state of the DxF DMA transfer.

Inputs: bCheckDRF – TRUE if wish to check the state of the DRF interface.

Outputs: RtnVal – no errors, busy, pending, page fault.

Prototype:

USHORT LynxHALCheckDMAState(BOOL bCheckDRF);

7.7.6 LynxHALDMABIPktRWSetup

Description: This function sets up the DMA hardware for transfer through the DxF data interface. The HW DMA control registers will be loaded with the information provided in the structure. The DMA mode should be the same as that used in the call to LynxHALDMASetup. The DMA transfer will NOT be started until you call LynxHALDMABIPktRWExec.

Inputs: RDop - 1 if 1394 READ operation or using the DRF to receive data, 0 if 1394 WRITE operation
*sDxFstruct - DxF control structure

Outputs: 0 - No errors. Otherwise, error.

Prototype:

```
#define LynxHALDMABIPktWrSetup( DxFstruct )
                                LynxHALDMABIPktRWSetup( 0, DxFstruct )
#define LynxHALDMABIPktRdSetup( DxFstruct )
                                LynxHALDMABIPktRWSetup( 1, DxFstruct )
```

```
USHORT LynxHALDMABIPktRWSetup (  BOOL Rdop,
                    LYNXHAL_DTF_DRF_CONTROL *sDTFControl );
```

7.7.7 LynxHALDMABIPktRWExec

Description: This function starts a 1394 PACKETIZED READ/WRITE operation to a remote node. You MUST call LynxHALDMABIPktRWSetup to setup the hardware for DMA transfer through the bulky data interface using the DRF/DTF.

Inputs: RDop - 1 if 1394 READ operation or using the DRF to receive data, 0 if 1394 WRITE operation
InitStart - 1 if INITIAL start of DMA engine, 0 if CONTINUEING a TIMEDOUT DMA operation

Outputs: None

Prototype:

```
#define LynxHALDMABIPktRdExec() LynxHALDMABIPktRWExec( 1, 1 )
#define LynxHALDMABIPktRdCont() LynxHALDMABIPktRWExec( 1, 0 )

#define LynxHALDMABIPktWrExec() LynxHALDMABIPktRWExec( 0, 1 )
#define LynxHALDMABIPktWrCont() LynxHALDMABIPktRWExec( 0, 0 )
void LynxHALDMABIPktRWExec( BOOL RDop, BOOL InitStart );
```

7.7.8 LynxHALDMABIPktRWRqst

Description: This function performs a 1394 PACKETIZED READ/WRITE operation to a remote node. It sets up the hardware for DMA transfer through the bulky data interface using the DRF/DTF an starts it.
NOTE: LYNXHALCheckDMAState is called, if an error state it is returned, else the DMA is setup and started. LynxHALAbortDmaWrite() should be called to clear errors.

Inputs: RDop - 1 if 1394 READ operation or using the DRF to receive data, 0 if 1394 WRITE operation
*sDxFstruct - DxF control structure

Outputs: None

Prototype:

```
#define LynxHALDMABIPktWrite( DxFstruct )
                                    LynxHALDMABIPktRWRqst( 0, DxFstruct )
#define LynxHALDMABIPktRead( DxFstruct )
                                    LynxHALDMABIPktRWRqst( 1, DxFstruct )

USHORT LynxHALDMABIPktRWRqst( BOOL RDop,
                        LYNXHAL_DTF_DRF_CONTROL *sDxFControl );
```

7.7.9 LynxHALAbortDma

Description: This function performs an ABORT of a 1394 PACKETIZED READ or WRITE operation to a remote node.

Inputs: RDop - 1 if 1394 READ operation, 0 if 1394 WRITE operation

Outputs: RtnVal – 0 if no errors

Prototype:

```
#define LynxHALAbortDmaWrite()  LynxHALAbortDma( 0 )
#define LynxHALAbortDmaRead()   LynxHALAbortDma( 1 )

USHORT LynxHALAbortDma( BOOL RDop );
```

7.8 SBP-2 Prototypes

The following sub-sections describe the SBP-2 interface functions provided in the TSB43AA82 HAL. The functions allow the upper layer to use the hardware support for SBP-2 that is available. There is no protocol support for SBP-2 provided by the firmware and very little provided by the hardware, it is up to the upper layers to complete the protocol requirements for SBP-2.

7.8.1 LynxHALClearMgmtAgentBusy

Description: This function clears the management agent busy bit when firmware is ready for hardware to receive the next management ORB.

Inputs: None

Outputs: None

Prototype:

void LynxHALClearMgmtAgentBusy(void);

7.8.2 LynxHALGetFreeAgentNumber

Description: This function returns the first available command agent number.

Inputs: None

Outputs: RtnVal - First available (0 to 3) command agent number or -1 if none available.

Prototype:

USHORT LynxHALGetFreeAgentNumber(void);

7.8.3 LynxHALValidCmdAgent

Description: This function assigns a command agent to a specific node ID.

Inputs: usAgtNum - Agent number (0 to 3) to assign
usNodeID - Node ID to assign

Outputs: RtnVal - TRUE if failed, FALSE if successful.

Prototype:

BOOL LynxHALValidCmdAgent(USHORT usAgtNum, USHORT usNodeID);

7.8.4 LynxHALGetAgentStatus

Description: This function checks command agent valid bit.

Inputs: usAgtNum - Agent number (0 to 3) to be checked

Outputs: RtnVal - TRUE if Agent valid, FALSE if Agent not valid

Prototype:

BOOL LynxHALGetAgentStatus(USHORT usAgtNum);

7.8.5 LynxHALClearCmdAgent

Description: This function clears the specified command agent valid bit.

Inputs: usAgtNum - Agent number (0 to 3) to clear

Outputs: RtnVal - TRUE if failed, FALSE if successful

Prototype:

BOOL LynxHALClearCmdAgent(USHORT usAgtNum);

7.8.6 LynxHALGetCommandAgentOffset

Description: This function returns the 1394 agent offset for the given agent number.

Inputs: usAgtNum - Agent number (0 to 3)

Outputs: RtnVal - Command agent offset low address, offset hi is always 0xFFFF

Prototype:

ULONG LynxHALGetCommandAgentOffset(USHORT usAgtNum);

7.8.7 LynxHALSetDead

Description: This function sets the DEAD bit in the agent status register.

Inputs: usAgtNum - Agent number (0 to 3) to assign

Outputs: RtnVal - TRUE if failed, FALSE if successful

Prototype:

BOOL LynxHALSetDead(USHORT usAgentNum);

7.8.8 LynxHALSetReset

Description: This function sets the RESET bit in the agent status register.

Inputs: usAgtNum - Agent number (0 to 3) to assign

Outputs: RtnVal - TRUE if failed, FALSE if successful

Prototype:

BOOL LynxHALSetReset(USHORT usAgentNum);

7.8.9 LynxHALSetAgentReady

Description: This function sets the agent ready bit for given agent number. The Agent Ready bit is what forces the hardware to fetch the next command ORB.

Inputs: usAgtNum - Agent number (0 to 3) to be set.

Outputs: None

Prototype:

void LynxHALSetAgentReady(USHORT usAgentNum);

7.8.10 LynxHALGetAgentState

Description: This function returns the information in the agent status register.

Inputs: usAgtNum - Agent number (0 to 3) to query the status

Outputs: RtnVal -

bit(s)	definition
0:3	Reserved
4:5	0: Reset, 1: Active, 2: Suspended, 3: Dead
6	Doorbell Rung
7	Unsolicited Status Enabled

Prototype:

UCHAR LynxHALGetAgentState(USHORT usAgentNum);

7.8.11 LynxHALClearDoorbell

Description: This function clears the doorbell bit for the given agent.

Inputs: usAgtNum - Agent number (0 to 3) to clear

Outputs: RtnVal - TRUE if failed, FALSE if successful

Prototype:

BOOL LynxHALClearDoorbell(USHORT usAgentNum);

7.8.12 LynxHALClearUnsolicitedStatus

Description: This function clears the unsolicited status bit for the given agent.

Inputs: usAgtNum - Agent number (0 to 3) to clear

Outputs: RtnVal - TRUE if failed, FALSE if successful

Prototype:

BOOL LynxHALClearUnsolicitedStatus(USHORT usAgentNum);

8. Installation

The firmware for the TSB43AA82 is supplied as an executable package and will be installed into a *<Targetdir>* directory of the user's choice that will have the following structure:

<Targetdir>

- Contains sub-directories for code and documentation as well as KEIL project files.
- <APIdir> - Contains source code for the API.
- <APPdir> - Contains source code for the Test Application.
- <DOCdir> - Contains this document as well as other related documents.
- <HALdir> - Contains the source code for the HAL.
- <HDDdir> - Contains the source code for the HDD application.
- <INCdir> - Contains the global include files needed by an application.
- <SYSdir> - Contains source code for "OS" type functionality.

Once the firmware has been installed, the user needs to make any modifications necessary to compile the code using his compiler of choice.

9. Porting Issues

Because it is expected that the LynxSoft firmware will be used in any number of different hardware and software environments, software portability was a consideration during its development. Some portability issues that an embedded system developer should consider are listed below.

9.1.1 Interface to Hardware

The developer should understand the Link Layer Controller hardware interface with the target system. The file LynxReg.h has the macros, and LynxReg.c has the functions, used to read and write Lynx hardware registers. These macros and functions should be optimized for the target system.

9.1.2 Operating System and ANSI C

LynxSoft firmware was written in ANSI C and without an operating system. If an OS will be used in the target system, the developer needs to consider the most efficient use of the OS with the LynxSoft design. How the OS handles interrupts, queued processes, and the time line to use these OS features are some of the things to be considered.

9.1.3 Interrupts and Callbacks

Whether an OS will be used or not, the developer needs to consider how interrupts and callback functions will be handled by the target system.

9.1.4 Memory usage.

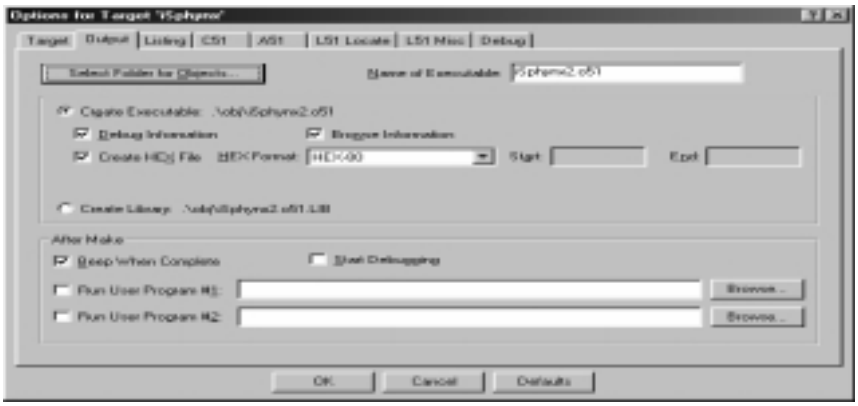
Because HAL layer software does not know the memory limitation of the target system, the HAL expects the upper layers and calling functions to provide all run time memory space. The calling

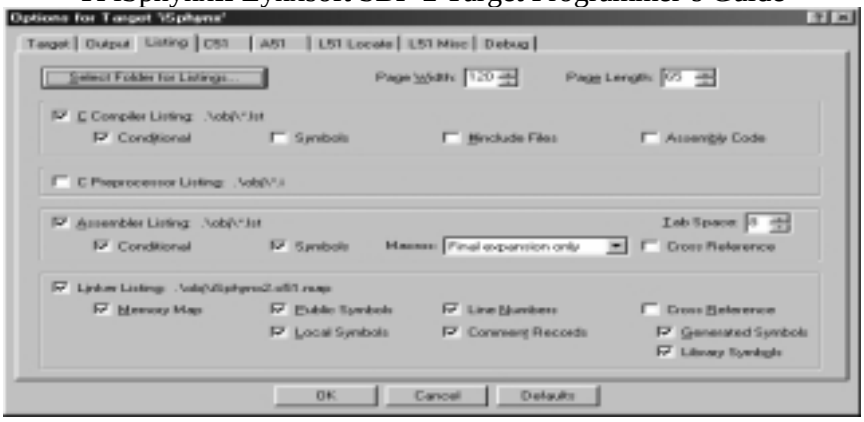
function needs to create and return the structures used for sending async packets, and create and return the space needed for incoming async packets.

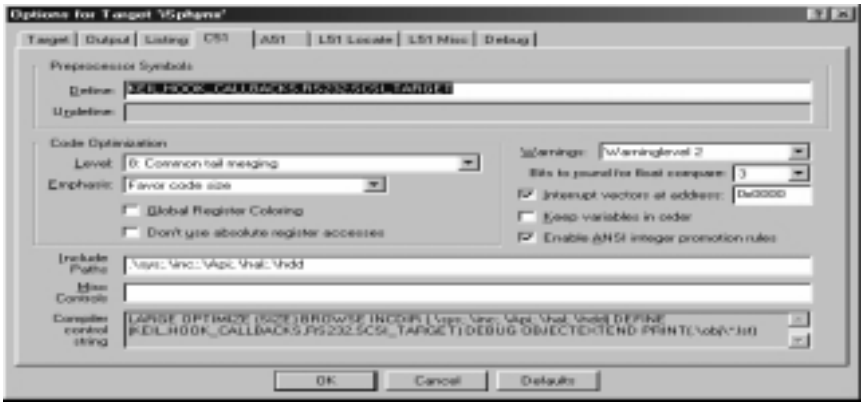
10. Tool Environments

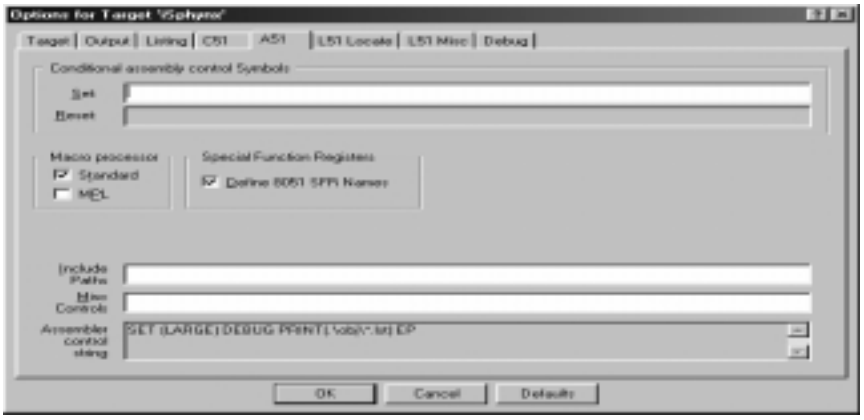
10.1 KEIL Development Environment

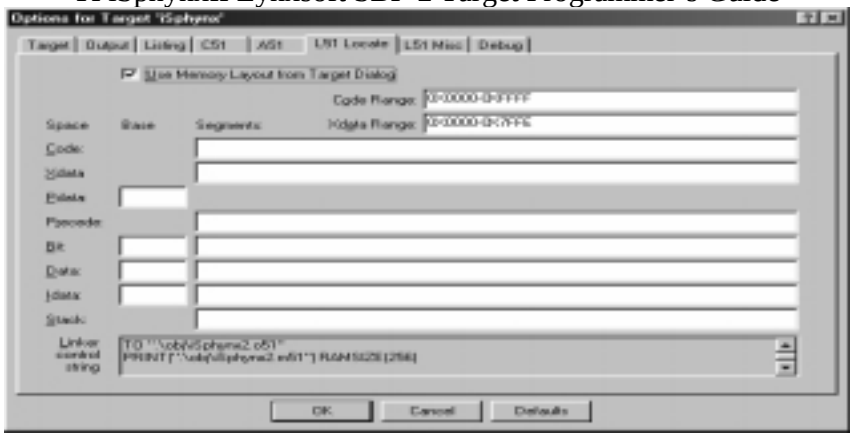
The following screenshots show the "project" options for the KEIL IDE, with which the firmware was compiled for the TI iSphynxII hardware reference platform.

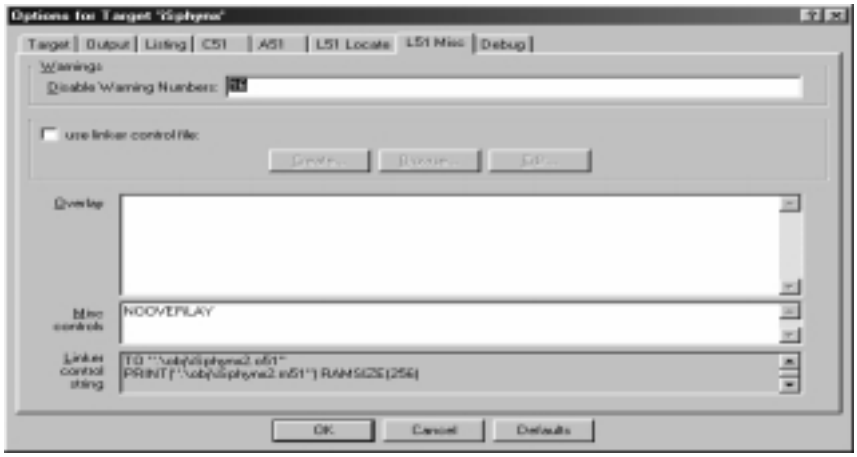


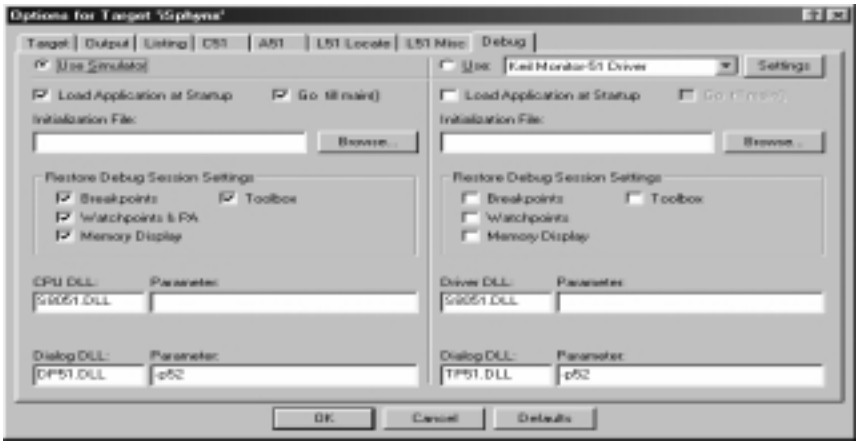

Options for Target 'iSphynx'
Target
Output
Listing
C51
A51
L51 Locate
L51 Misc
Debug
Select Folder for Listings...
Page Width:
Page Length:
C Compiler Listing: .\obj*.lst
Conditional
Symbols
#include Files
Assembly Code
C Preprocessor Listing: .\obj*.i
Assembler Listing: .\obj*.lst
Tab Space:
Conditional
Symbols
Macros:
Final expansion only
Cross Reference
Memory Map
Public Symbols
Local Symbols
Line Numbers
Comment Records
Cross Reference
Generated Symbols
Library Symbols
OK
Cancel
Defaults


Options for Target 'iSphynx'
Target
Output
Listing
C51
A51
L51 Locate
L51 Misc
Debug
Preprocessor Symbols
Define:
Undefine:
Code Optimization
Level:
0: Common tail merging
Emphasis:
Favor code size
Global Register Coloring
Don't use absolute register accesses
Warnings:
Warninglevel 2
Bits to round for float compare:
Interrupt vectors at address:
0x0000
Keep variables in order
Enable ANSI integer promotion rules
Include Paths
\sys;..\inc;..\Api;..\hal;..\hdd
Misc Controls
Compiler control string
OK
Cancel
Defaults


Options for Target 'iSphynx'
Target
Output
Listing
C51
A51
L51 Locate
L51 Misc
Debug
Conditional assembly control Symbols
Set
Reset
Macro processor
Standard
MPL
Special Function Registers
Define 8051 SFR Names
Include Paths
Misc Controls
Assembler control string
SET (LARGE) DEBUG PRINT(.\obj*.lst) EP
OK
Cancel
Defaults

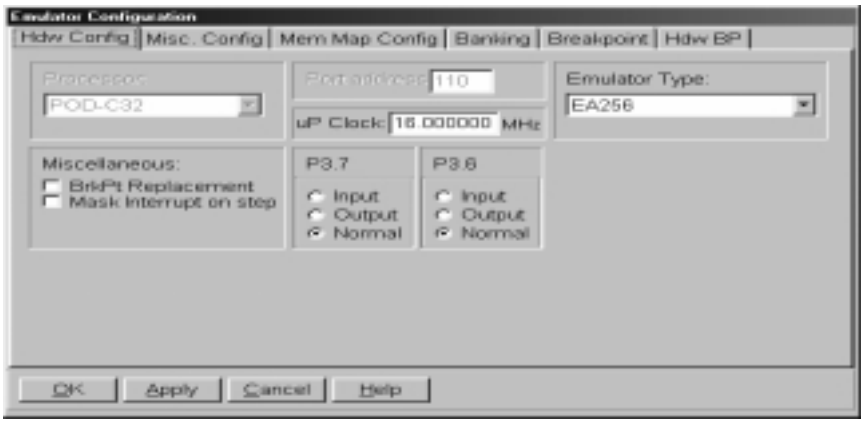

Options for Target 'iSphynx'
Target
Output
Listing
C51
A51
L51 Locate
L51 Misc
Debug
Use Memory Layout from Target Dialog
Code Range:
Space
Base
Segments:
Xdata Range:
Code:
Xdata
Pdata
Pwords:
Bit:
Data:
Idata:
Stack:
Linker control string
OK
Cancel
Defaults

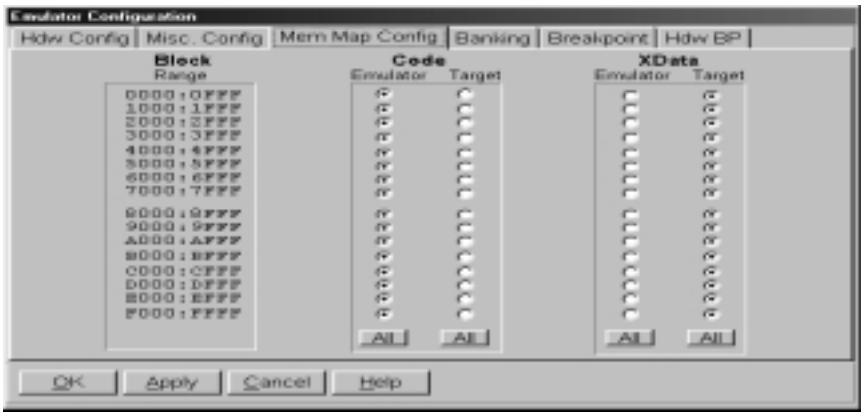

Options for Target 'iSphynx'
Target
Output
Listing
C51
A51
L51 Locate
L51 Misc
Debug
Warnings
Disable Warning Numbers:
use linker control file:
Overlay
Misc controls
NOOVERLAY
Linker control string
OK
Cancel
Defaults

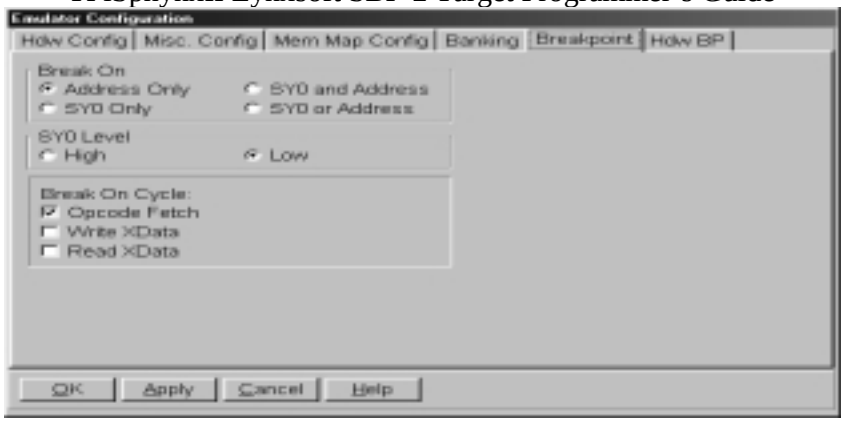

Options for Target 'iSphynx'
Target
Output
Listing
C51
A51
L51 Locate
L51 Misc
Debug
Use Simulator
Load Application at Startup
Go till main()
Initialization File:
Browse...
Restore Debug Session Settings
Breakpoints
Toolbox
Watchpoints & PA
Memory Display
CPU DLL:
Parameter:
S8051.DLL
Dialog DLL:
DP51.DLL
p52
Use:
Keil Monitor-51 Driver
Settings
Load Application at Startup
Driver DLL:
TP51.DLL
OK
Cancel
Defaults

10.2 NOHAU Environment

The following screenshots show the "Configuration" options for the NOHAU debugger used in conjunction with the TI iSphynxII hardware reference platform.






Emulator Configuration
Hdw Config
Misc. Config
Mem Map Config
Banking
Breakpoint
Hdw BP
Break On
Address Only
SY0 and Address
SY0 Only
SY0 or Address
SY0 Level
High
Low
Break On Cycle:
Opcode Fetch
Write XData
Read XData
OK
Apply
Cancel
Help

Appendix A

The following is a list of status and error codes used throughout the API:

0x0000 SBI_NO_ERROR – No errors.

0x0202 SBI_ERROR_PACKET_SIZE_TOO_BIG – Packet size is too large.
0x0203 SBI_ERROR_PACKET_INFO_BAD – Contents of packet are bad.
0x0204 SBI_ERROR_SEND_PACKET_QUEUE_FULL – Send queue is full.
0x0205 SBI_ERROR_SEND_PACKET_RETRY_MAX – Reached maximum retries.
0x0206 SBI_ERROR_SEND_PACKET_INVALID_ACK – Ack of sent packet is invalid.
0x0207 SBI_ERROR_SEND_PACKET_BUS_RESET – Reset occurred before completed transaction.
0x0208 SBI_ERROR_SEND_RESPONSE_TIMEOUT – Timeout waiting for response.
0x0209 SBI_ERROR_SEND_IO_TIMEOUT – Timeout waiting for IO to complete.
0x0210 SBI_ERROR_RANGE_LIST_MAX – Reached max number of allocations.
0x0211 SBI_ERROR_RANGE_COULD_NOT_FREE – Could not free address range.
0x0212 SBI_ERROR_RANGE_OVERLAP – Overlap of other range or space.
0x0213 SBI_ERROR_RANGE_SIZE – Past max allowed range size.
0x0214 SBI_ERROR_RANGE_ADDRESS_LIMIT – Exceeded address limit.

0x0231 SBI_ERROR_CONFIG_ROM_LENGTH – Invalid length of Config ROM.

0x0250 SBI_ERROR_IN_DEVELOPMENT – Currently under development.
0x0251 SBI_ERROR_PARAMETERS – Parameter inputs improper or in conflict

0x0261 SBI_ERROR_MEMORY – Failed dynamic memory allocation