ESP-488TL

Software Reference Manual for TNT4882™

July 1996 Edition

Part Number 370874A-01

© Copyright 1993, 1994 National Instruments Corporation.
All Rights Reserved.

Internet Support

GPIB: `gpib.support@natinst.com`
DAQ: `daq.support@natinst.com`
VXI: `vxi.support@natinst.com`
LabVIEW: `lv.support@natinst.com`
LabWindows: `lw.support@natinst.com`
HiQ: `hiq.support@natinst.com`
VISA: `visa.support@natinst.com`

E-mail: `info@natinst.com`
FTP Site: `ftp.natinst.com`
Web Address: `http://www.natinst.com`

Bulletin Board Support

BBS United States: (512) 794-5422 or (800) 327-3077
BBS United Kingdom: 01635 551422
BBS France: 1 48 65 15 59

FaxBack Support

(512) 418-1111

Telephone Support (U.S.)

Tel: (512) 795-8248
Fax: (512) 794-5678

International Offices

Australia 03 9 879 9422, Austria 0662 45 79 90 0, Belgium 02 757 00 20,
Canada (Ontario) 519 622 9310, Canada (Québec) 514 694 8521,
Denmark 45 76 26 00, Finland 90 527 2321, France 1 48 14 24 24,
Germany 089 741 31 30, Hong Kong 2645 3186, Italy 02 413091, Japan 03 5472 2970,
Korea 02 596 7456, Mexico 95 800 010 0793, Netherlands 0348 433466,
Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085,
Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200,
U.K. 01635 523545

National Instruments Corporate Headquarters

6504 Bridge Point Parkway Austin, TX 78730-5039 Tel: (512) 794-0100

Limited Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this manual is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

NAT4882®, Turbo488®, NI-488.2M™, and TNT4882™ are trademarks of National Instruments Corporation.

Product and company names listed are trademarks or trade names of their respective companies.

Warning Regarding Medical and Clinical Use of National Instruments Products

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

Contents

About This Manual xi
Organization of This Manual xi
Conventions Used in This Manual xii
Related Documentation xiii
Customer Communication xiii

Chapter 1
Introduction 1-1
What Your Kit Should Contain 1-1
File Description 1-1
Important Considerations 1-2
Programming Considerations 1-2
ESP Code Considerations 1-3

Chapter 2
The GPIB C Language Library 2-1
Global Variables 2-1
Status Variable: INTERFACE_STATUS 2-1
Error Variable: INTERFACE_ERROR 2-5
Count Variable: DATA_COUNT 2-6
Address Variables: CURRENT_ADDRESS and MULTI_ADDRESS 2-6
Read and Write Termination 2-6
Compiling a C Device Program with the Noninterrupt ESP Package 2-7
Compiling a C Device Program with the Interrupt ESP Package 2-8
GPIB Function Descriptions 2-8
Status Functions 2-9
Initialization Functions 2-9
Addressing Functions 2-10
I/O Functions 2-10
Interrupt Control Functions 2-11

Chapter 3
TNT4882 Talker/Listener ESP Functions and Utilities Reference 3-1
void Abort_ASYNC_IO(void) 3-2
void Change_Primary_Address(int new_address) 3-3
void Change_Multiple_Addresses(int *address_list, int num_of_addresses) 3-4
void Change_Secondary_Address(int new_address) 3-6
void Clear_4882_Status(int reg, int mask) 3-7
void Clear_Interrupt_Function(int reg, int mask, void (*function)()) 3-8
void High_Speed_Select(int cable_length, int enable) 3-9

void Initialize_Interface() 3-10
void Interface_Off() 3-11
int Read_4882_Status(int reg) 3-12
int Read_GPIB_Lines() 3-13
void Receive(char *buf,unsigned long int cnt,int term) 3-14
void Receive_ASYNC(char *buf,unsigned long int cnt,int term) 3-16
void Send(char_buf *buf, unsigned long cnt, int term) 3-18
void Send_ASYNC(char_buf *buf, unsigned long cnt, int term) 3-20
void Set_4882_Status(int reg, int mask) 3-22
void Set_Address_Mode(int mode) 3-23
void Set_Interrupt_Function(int reg, int mask, void (*function)()) 3-25
void Set_Timeout(int factor_index, int byte_timeouts) 3-27
int Update_INTERFACE_STATUS() 3-29
void Wait_For_Interface(int mask) 3-31

Chapter 4
Queues 4-1
Queue Models 4-2
Structure of a Queue 4-3
Queue Information Globals 4-4
QUEUE_STATUS 4-5
Count Variable: QUEUE_COUNT 4-7
Address Variable: QUEUE_ADDRESS 4-7
Configuring the Queues 4-8
Queue Operation 4-8
Input Queue Handling 4-9
Input Queue and Events 4-9
Output Queue Handling 4-9
Queue Function Descriptions 4-10
Initialization Functions 4-10
Status Functions 4-10
I/O Functions 4-10

Chapter 5
TNT4882 Talker/Listener ESP Queue Functions and Utilities Reference 5-1
int Data_Available() 5-2
void Disable_Input_Queue() 5-3
void Disable_Output_Queue() 5-4
void Enable_Input_Queue() 5-5
void Enable_Output_Queue() 5-6
void Initialize_Input_Queue() 5-7
void Initialize_Output_Queue() 5-8
int Message_Available() 5-9
unsigned long int Message_Length() 5-10
unsigned long int Output_Space_Available() 5-11
void Read_Input_Queue(char_buf *buf, unsigned long int cnt, int *term) 5-13
void Wait_For_Queue(int mask) 5-15

void Write_Output_Queue(char_buf *buf, unsigned long int cnt, int term) 5-17

Chapter 6
Advanced Topics 6-1
TNT4882 Initialization 6-1
Initialization Steps 6-1
Configuring the TNT4882 Handshake Mode and High-Speed I/O 6-3
Detecting the CF Command 6-3
Configuring the TNT4882 Mode and Timing Registers 6-4
Setting the TNT4882 to the Normal Three-Wire Handshake Mode 6-5
Setting the TNT4882 to HS488 6-5
Addressing Modes and Detecting the Address State 6-6
Single Primary Addressing 6-6
Configuring for Single Primary Addressing 6-6
Single Primary–Single Secondary Addressing 6-7
Configuring for Single Primary–Single Secondary Addressing 6-7
Single Primary–Multiple Secondary Addressing 6-7
Configuring for Single Primary–Multiple Secondary Addressing 6-8
Listen-Only/Talk-Only Addressing 6-8
Configuring for Talk-Only/Listen-Only Addressing 6-9
Talk Only 6-9
Listen Only 6-9
Multiple Primary Addressing 6-9
Configuring for Multiple Primary Addressing 6-9
Multiple Primary and Multiple Secondary Addressing 6-10
Configuring for Multiple Primary and Multiple Secondary Addressing 6-10
GPIB I/O with the TNT4882 6-11
Setting Up the TNT4882 for I/O: Setup_TNT_IO() 6-11
Set Up the TNT4882 to Input Data 6-12
Set Up the TNT4882 to Output Data 6-13
Interrupt and Noninterrupt Programmed I/O: GPIB_PROG_IO() 6-14
Important I/O Status Bits 6-14
Handling Noninterrupt Programmed I/O 6-14
Inputting Data 6-14
Outputting Data 6-15
Handling Interrupt-Driven Programmed I/O 6-16
Inputting Data 6-16
Outputting Data 6-17
Interrupt and Noninterrupt DMA I/O: GPIB_DMA_IO() 6-17
Handling Noninterrupt DMA I/O with the Intel 8237 DMA Controller 6-18

Inputting and Outputting Data 6-18
Handling Interrupt-Driven DMA I/O 6-19
Cleaning Up I/O: DONE_Handler() &
DONE_Interrupt_Handler() 6-20
Noninterrupt and Interrupt Clean Up 6-20
Optimizing I/O.. 6-21
I/O Testing 6-22
Eliminate Loops 6-22
Eliminate Unnecessary Tasks 6-22
Do Not Use Interrupts 6-23
Compiling: Check the Assembled Code 6-24
Interrupts and the TNT4882 ESP-488TL (inter_io.c) 6-24
Enabling TNT4882 Interrupts 6-24
Initially Enabling Interrupt Handler 6-24
Enabling TNT4882 Interrupts For IMR0, IMR1, and
IMR2 6-25
Enabling an ISR3 Interrupt to Occur 6-25
Detection of the Interrupt Condition 6-25
Detecting an Interrupt Condition 6-25
Clearing an Interrupt Condition 6-26
Three Methods of Clearing Interrupt
Conditions.. 6-26
How the Interrupt Handler() Calls an Interrupt
Routine 6-26
Event Handling 6-27
Clearing the INTERFACE_STATUS Bits 6-27
Including Events in the Input Queue and Creating New Interrupt
Event Handlers 6-28
Multichip Communications 6-29
IEEE 488.2 Status Model.. 6-30
Required IEEE 488.2 Status Registers 6-31
Adding New Registers to the Status Model 6-31
ESP Coding Conventions.. 6-32
Helpful Source Code Rules 6-32

Appendix A
Configuring the TNT4882 Talker/Listener ESP A-1
Configuring the Noninterrupt and Interrupt ESP... A-1
Addressing Parameters A-2
I/O Parameters.. A-3
Miscellaneous Handlers... A-4
Hardware Parameters.. A-6
Software Configuration A-7
Queue Model Parameters A-7

Appendix B
Programming Examples B-1
Basic Device Program Model B-1
Interrupt ESP Examples B-2
Noninterrupt ESP Examples B-9

Appendix C
Mnemonics Key C-1

Appendix D
Customer Communication D-1

Glossary G-1

Index I-1

Figures

Figure 4-1. Queue Models 4-2
Figure 4-2. Queue Structure 4-3

Tables

Table 1-1. Design Considerations 1-3

Table 2-1. TNT4882 Interface Status Word Bits 2-2
Table 2-2. INTERFACE_ERROR Descriptions 2-5

Table 3-1. Waiting Time 3-27
Table 3-2. INTERFACE_STATUS Status Word 3-29

Table 4-1. QUEUE_STATUS Status Bits and Bit Positions 4-5
Table 4-2. io_param.h Queue Configuration Constants 4-8

Table 5-1. QUEUE_STATUS Status Bits 5-15

Table 6-1. Timing Registers Configuration Table 6-4

Table A-1. Addressing Parameters A-2
Table A-2. I/O Parameters A-3
Table A-3. Miscellaneous Handlers A-4
Table A-4 TNT4882, DMA, and Interrupt Hardware Parameters A-6
Table A-5. Software Configuration A-7
Table A-6. Queue Model Parameters (For the Interrupt ESP Only) A-7

About This Manual

The *ESP-488TL Software Reference Manual for TNT4882* describes the TNT4882 Talker/Listener (TL) software and advanced TNT4882 programming information.

This manual assumes that you are familiar with the IEEE 488 GPIB standard and the IEEE 488.2 GPIB messaging protocols. Use the IEEE 488 GPIB standard and the IEEE 488.2 GPIB messaging protocols when you develop a 488-compliant device.

Organization of This Manual

This manual is organized as follows:

- Chapter 1, *Introduction,* reviews the TNT4882 ESP-488TL source code kit and discusses the contents of the source code files. It also reviews the files on the ESP distribution diskette and describes important programming considerations.
- Chapter 2, *The GPIB C Language Library*, contains a general description of the C language programming interface to the TNT4882 evaluation board and a brief description of the TNT4882 ESP-488TL interrupt and noninterrupt package functions.
- Chapter 3, *TNT4882 Talker/Listener ESP Functions and Utilities Reference*, explains how to use the functions and utilities in the TNT4882 ESP-488TL package. This chapter lists the functions in alphabetical order.
- Chapter 4, *Queues*, discusses the TNT4882 ESP-488TL I/O queues.
- Chapter 5, *TNT4882 Talker/Listener ESP Queue Functions and Utilities Reference,* contains information that explains how to use the queue functions and utilities in the TNT4882 ESP-488TL interrupt package. This chapter lists the functions in alphabetical order.
- Chapter 6, *Advanced Topics*, explains TNT4882 programming fundamentals and issues that you need to be aware of when you rewrite or change the ESP code.
- Appendix A, *Configuring the TNT4882 Talker/Listener ESP*, explains how to configure the interrupt and noninterrupt ESP.
- Appendix B, *Programming Examples*, provides six programming examples that demonstrate various I/O models and how to use some of the ESP function calls.
- Appendix C, *Mnemonics Key*, defines the mnemonics (abbreviations) that this manual uses for functions, remote messages, local messages, states, bits, registers, integrated circuits, and system functions.

- Appendix D, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and a description of the terms, such as abbreviations, acronyms, metric prefixes, mnemonics, and symbols, that this manual uses.
- The *Index* contains an alphabetical list of the key terms and topics that this manual uses, and it includes the page number where you can locate each term and topic.

Conventions Used in This Manual

This manual uses the following conventions.

italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept.
bold italic	Bold italic text denotes a note, caution, or warning.
`monospace`	Lowercase text in this font denotes text or characters that you literally input from the keyboard. It also denotes sections of code, programming examples, syntax examples, the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, variables, filenames, extensions, parameters, and statements and comments taken from program code.
`italic monospace`	Italic lowercase text in this font denotes that you must supply the appropriate words or values in the place of these items.
IEEE 488 and IEEE 488.2	IEEE 488 and IEEE 488.2 refer to the ANSI/IEEE Standard 488.1-1987 and the ANSI/IEEE Standard 488.2-1987, respectively, which define the GPIB.

The *Glossary* lists abbreviations, acronyms, metric prefixes, mnemonics, symbols, and terms.

Related Documentation

The following documents contain information that you may find helpful as you read this manual.

- ANSI/IEEE Standard 488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation*
- ANSI/IEEE Standard 488.2-1987, *IEEE Standard Codes, Formats, Protocols, and Common Commands*

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix D, *Customer Communication*, at the end of this manual.

Chapter 1
Introduction

This chapter reviews the TNT4882 ESP-488TL source code kit and discusses the contents of the source code files. It also reviews the files on the ESP (Engineering Software Package) distribution diskette and describes important programming considerations.

The TNT4882 ESP-488TL development software helps the GPIB developer program the National Instruments TNT4882 chip by providing source code that performs the necessary 488 device functions. The TNT4882 ESP-488TL code contains functions for synchronous and asynchronous device communications, 488 status reporting, asynchronous I/O and event queues, high-speed data transfer configuration, and device configuration.

What Your Kit Should Contain

Your kit should contain the following components:

- DOS-formatted, high-density diskette containing TNT4882 ESP-488TL (part number 413118-103)
- *ESP-488TL Software Reference Manual for TNT4882* (part number 320725-01)

File Description

Check your ESP distribution diskette for the following files:

`ints`	Interrupt-Driven Source Directory
`io_param.h`	Device Configuration Parameter Header File
`gpib_io.c`	Interrupt-Driven GPIB Source Code
`gpib_io.h`	Interrupt-Driven GPIB Header File
`inter_io.c`	Interrupt Control Functions Source Code
`inter_io.h`	Interrupt Control Functions Header File
`queue_io.c`	Queue Source Code
`queue_io.h`	Queue Header File
`ex1.c`	Programming Example 1
`ex2.c`	Programming Example 2
`ex3.c`	Programming Example 3
`ex4.c`	Programming Example 4
`noints`	Noninterrupt (Polled) Source Directory
`io_param.h`	Device Configuration Parameter Header File
`ngpib_io.c`	Noninterrupt GPIB Source Code
`ngpib_io.h`	Noninterrupt GPIB Header File
`ex5.c`	Programming Example 5
`ex6.c`	Programming Example 6

Important Considerations

Before you use the TNT4882 ESP-488TL software, you must install the TNT4882 evaluation board and load the ESP source code. Information about installing the evaluation board is in the *TNT4882 Evaluation Board Installation Guide* (National Instruments part number 320726-01). You should configure the software before you use it with the evaluation board. Refer to Appendix A, *Configuring the TNT4882 Talker/Listener ESP*, in this manual.

Consider the following information when developing a device program using the TNT4882 ESP-488TL.

Programming Considerations

- Two ESP packages are on the distribution diskette: one is interrupt driven and one is noninterrupt driven. It is easier for you to follow the noninterrupt-driven ESP (`ngpib_io.c`) when you begin programming.
- You must include the appropriate header files to describe the prototypes for functions called in the device program.
- The first function that must be called is `Initialize_Interface()`. This function sets up addressing modes and interrupt vectors (if they are necessary) and resets the TNT4882.
- The last GPIB function that should be called is `Interface_Off()`. Calling this function last prevents the TNT4882 from receiving any further GPIB data and commands.
- A GPIB Controller must appropriately address the device program to talk or listen before I/O functions can be implemented.
- The ESP contains I/O functions that can be synchronous or synchronous/asynchronous DMA or programmed I/O.
- Only the interrupt version of the TNT4882 ESP-488TL includes asynchronous I/O and interrupt routines.
- Three status variables—`INTERFACE_STATUS`, `DATA_COUNT`, and `INTERFACE_ERROR`—contain important information about the status of the function call last implemented.
- The ESP code was developed by using a TNT4882 evaluation board under DOS 5.0. The code is compatible with both Microsoft C and Borland C compilers. Programming optimizations have not been made, so the code is easier to read. These optimizations are described in the *Optimizing I/O* section in Chapter 6, *Advanced Topics*.

- You can find additional information about programming the TNT4882 in the *TNT4882 Initialization* section in Chapter 6, *Advanced Topics*, and in the *TNT4882 Programmer's Reference Manual* (National Instruments part number 320724-01).

ESP Code Considerations

Before using the ESP, you should consider your device design and what processor, DMA, and interrupt controllers you will use. Both ESP packages have distinct advantages. For instance, the noninterrupt package is usually faster once I/O begins, but the compiled size of the object modules is smaller in the noninterrupt package. The interrupt-driven package permits asynchronous I/O calls and the queuing of data messages and events. However, porting the interrupt-driven package to another processor and chipset is generally more difficult than porting the noninterrupt package, because the interrupt package contains code specific to the 8259 interrupt controller.

Table 1-1 summarizes ESP design specifications that can affect your ESP code choice.

Table 1-1. Design Considerations

Design Specifications	Considerations
Processor	Portability of Existing Code: Noninterrupt ESP is Easiest
Interrupt Controller	Interrupt Handler and Routines Written for 8259 Interrupt Controller
DMA Controller	DMA I/O Written for 8259 DMA Controller
Code Size	Noninterrupt (smallest) < Interrupt Driven (larger) < Interrupt Driven + Queues (largest)
Code Speed	I/O Fastest in Noninterrupt-Driven Package
Event Handling	Fastest in Interrupt-Driven Package
Basic Code	Noninterrupt-Driven Package

Before you use the interrupt-driven package, become familiar with the noninterrupt package: the functions are similar in both packages but easier to understand in a noninterrupt-driven environment. When you develop a device program, you may want to use the noninterrupt package to develop device I/O handling code and routines such as clearing and triggering devices, then convert these routines to the interrupt package.

Chapter 2
The GPIB C Language Library

This chapter contains a general description of the C language programming interface to the TNT4882 evaluation board and a brief description of the TNT4882 ESP-488TL interrupt and noninterrupt package functions.

Global Variables

When performing I/O operations by executing a Send/Receive function, the following global variables are used to describe the GPIB status, the error conditions, and the data transfer count of the TNT4882.

`INTERFACE_STATUS`	Describes the TNT4882 address states and event conditions.
`INTERFACE_ERROR`	Gives the error code but is not meaningful unless the ERR bit is set in `INTERFACE_STATUS`.
`DATA_COUNT`	Gives the number of bytes read or written over the GPIB. This number can be an unsigned integer value in the range of 0 to 4 GB.
`CURRENT_ADDRESS`	Is the current GPIB address of the TNT4882.
`MULTI_ADDRESS`	Indicates the addresses for sends and receives, but only when using multiple addressing options.

Status Variable: INTERFACE_STATUS

I/O functions automatically update the INTERFACE_STATUS status word, which contains 16 status bits. You can also use the ESP function Update_INTERFACE_STATUS() to update the INTERFACE_STATUS word to its latest value. Table 2-1 lists INTERFACE_STATUS status bits and their bit positions.

Table 2-1. TNT4882 Interface Status Word Bits

Mnemonic	Bit Position	Description
ERR	15	GPIB Error
TIMO	14	Time Limit Exceeded
END	13	End Detected
EOS	12	End-of-String Detected
RQS	11	Requesting Service
IFC	10	Interface Clear
SPOLL	9	Serial Poll Active
UCMPL	8	User I/O Complete
LOK	7	Local Lockout
REM	6	Remote Programming
ASYNC	5	Asynchronous I/O In Progress
DTAS	4	Device Trigger Active State
DCAS	3	Device Clear Active State
LACS	2	Listener Active State
TACS	1	Talker Active State
NACS	0	Not Active State

ERR

The ERR bit is set when a function does not complete because of I/O problems or improper arguments in the function call. Check INTERFACE_ERROR for the error type. ERR is cleared only when a new I/O call is executed. No error-handling routines are provided in the ESP, so you must clear the bit in all other cases. See the *Event Handling* section in Chapter 6, *Advanced Topics*.

TIMO — The TIMO bit is set if timeouts are used in the device program and the time for an I/O transfer to complete is expired. You can configure timeout time by using the `Set_Timeout()` function. TIMO is cleared by the initiation of a new I/O operation.

END — The END bit denotes that a reading operation received a terminator. This terminator can be the EOI GPIB signal and/or EOS terminator. END is cleared by the initiation of a new I/O operation.

EOS — The EOS bit denotes that the TNT4882 received an EOS byte as a terminator. EOS is cleared by the initiation of a new I/O operation.

RQS — The RQS bit indicates that the TNT4882 is currently requesting service from the GPIB Controller. RQS is cleared after it has been serial polled by the GPIB Controller.

IFC — The IFC bit indicates that a System Controller is asserting or has asserted the GPIB IFC signal. See the *Note* below.

SPOLL — The SPOLL bit indicates that the Controller is requesting a new serial poll byte. SPOLL is cleared when a new status byte is written to the Serial Poll Mode Register (SPMR). For the noninterrupt ESP, SPOLL can be enabled or disabled with the `USE_SPOLL_BIT` constant in the `io_param.h` configuration file. SPOLL is not used in the interrupt ESP. See the *Note* below.

UCMPL — The UCMPL bit indicates that a `Send()` or `Receive()` function initiated by the program has completed. UCMPL is cleared each time you begin a new I/O call. If you are performing asynchronous transfers, you can use UCMPL to detect when an I/O process completes and when another may begin. UCMPL is not set or cleared by the I/O queues.

LOK — The LOK bit indicates that the TNT4882 RL function is in a lockout state. See the IEEE 488.1 standard for more information on the RL function.

REM — The REM bit indicates if the TNT4882 is in remote state. REM is set when the Remote Enable (REN) GPIB signal is asserted and the TNT4882 detects its listen address. REM is cleared when REN becomes unasserted or the TNT4882 detects the Go To Local (GTL) command.

ASYNC — The ASYNC bit indicates that the device program has asynchronous I/O in progress. You should check ASYNC before initiating an ASYNC transfer. This bit is cleared when the ASYNC process terminates and is never set by the noninterrupt-driven ESP.

DTAS — The DTAS bit indicates that the TNT4882 received a Group Execute Trigger (GET) multiline interface command. See the *Note* below.

DCAS — The DCAS bit indicates that the TNT4882 received a Device Clear (DCL) or a Selected Device Clear (SDC) multiline interface command. See the *Note* below.

LACS — The LACS bit indicates that the TNT4882 received the listen address of the TNT4882. LACS is cleared when the interface receives its talk address or becomes unaddressed to listen by the Unlisten (UNL) multiline interface message.

TACS — The TACS bit indicates that the TNT4882 received the talk address of the interface. TACS is cleared when the interface receives its listen address or becomes unaddressed to talk by the Untalk (UNT) multiline interface message.

NACS — The NACS bit indicates that the TNT4882 is not addressed to talk or listen. NACS is set when the TACS and LACS bit is clear. NACS is cleared when the TACS or LACS bit is set.

Note: ***The ESP-488 packages do not contain complete functions for handling DCAS, DTAS, SPOLL, and IFC. The interrupt ESP contains only simple interrupt handlers for these events. These simple handlers can be enabled by the*** `USE_EVENT HANDLERS` ***configuration option. (The actions that you should take are described by the IEEE 488.2 standard and are device specific.) Therefore, you must clear these bits programmatically when you develop routines to handle these events. Refer to the*** *Event Handling* ***section in Chapter 6,*** *Advanced Topics*, ***for more information.***

Error Variable: INTERFACE_ERROR

When the ERR bit in the `INTERFACE_STATUS` status word is set, the error code from `INTERFACE_ERROR` is valid. Only I/O functions automatically clear the ERR bit in `INTERFACE_STATUS`. Errors listed in Table 2-2, with the exception of EARG, are I/O errors.

Table 2-2. INTERFACE_ERROR Descriptions

Mnemonic	Value	Description
—	0	No Errors
ENOL	1	No Listeners—I/O Aborted Because No Listeners on Bus
EARG	2	Bad Argument in Parameter List
EABO	3	I/O Aborted Due to Timeout
EOIP	4	Asynchronous Operation in Progress
EIQ	5	Input Queue Error (Check `QUEUE_STATUS`)
EOQ	6	Output Queue Error (Check `QUEUE_STATUS`)
EAQ	7	Active Queue—Warning Not to Initiate Other I/O Calls While Input or Output Queue Performs I/O

ENOL
: The ENOL code indicates that the TNT4882 tried to output data, but no device asserted the GPIB Not Data Accepted (NDAC) signal.

EARG
: The EARG code indicates that an improper argument has been passed to a function and that correct operation is not assured.

EABO
: The EABO code is caused by either an I/O timeout or a termination of asynchronous I/O. Timeout time can be increased, decreased, or completely disabled by the `Set_Timeout()` function.

EOIP
: The EOIP code indicates that an asynchronous I/O process is underway and that no other I/O is allowed until it completes or aborts.

EIQ
: The EIQ code indicates that either no data is in the Input Queue when a `Read_Input_Queue()` function is executed or an event such as Device Clear (DEC) has been lost because the Input Queue has no room for a new message. EIQ can also indicate a memory allocation error.

EOQ — The EOQ code indicates that no new data can be placed in the Output Queue because it is full. EOQ can also indicate a memory allocation error.

EAQ — The EAQ code indicates that you cannot initiate an I/O call independently of the queues because a queue has currently begun an asynchronous I/O process.

Count Variable: DATA_COUNT

The `DATA_COUNT` variable is updated with the actual byte count that is transferred over the GPIB after every I/O function has completed. Asynchronous I/O is valid only after the asynchronous I/O process terminates.

Address Variables: CURRENT_ADDRESS and MULTI_ADDRESS

`CURRENT_ADDRESS` reflects the current address of the TNT4882 chip. This word is made up of a primary address (the upper byte) and a secondary address (the lower byte). This value can change if you use the multiple addresses options in the `io_param.h` configuration file.

If using multiple addressing options, `MULTI_ADDRESS` indicates which listen address the TNT4882 received data at. `MULTI_ADDRESS` can also configure the `Send()` function to write from a specific talk address. See the Send/Receive functions for more information.

Read and Write Termination

The IEEE 488.1 standard defines two methods of identifying the last byte of device-dependent messages. Both methods let a Talker send data messages of any length without the Listener(s) knowing the number of bytes in the transmission in advance. The two methods are the following:

- END message — The Talker simultaneously asserts the EOI signal with the transmission of the last data byte.
- EOS character — The Talker transmits a special data byte that the Listener interprets as the END message.

You can configure the two `io_param.h` constants, `READ_EOS_BYTE` and `WRITE_EOS_BYTE`, to determine what kind of EOS byte the TNT4882 will recognize for EOS termination.

- `READ_EOS_BYTE` When reading, this constant is used to configure the EOS termination byte of the TNT4882. Typically, this byte is the ASCII linefeed character (0x0a) for IEEE 488.2 devices.

- `WRITE_EOS_BYTE` When writing, this constant is used to configure the TNT4882 to set the EOI GPIB signal and the transmission of the EOS byte. This byte is typically the ASCII linefeed character (0x0a).

The I/O function calls do not use these EOS bytes unless the termination parameter (`term`) in the I/O function calls is set to EOS or EOI|EOS.

Compiling a C Device Program with the Noninterrupt ESP Package

In addition to any other necessary header file for the device program, you should include the following header files for the noninterrupt-driven package:

```
#include "io_param.h"
#include "ngpib_io.h"
```

You should include the following compiling options when you compile the device code:

`/AH, -mh`	for the Huge memory model
`/Ot, -O2`	to optimize for execution speed (optional)

Compiling with the Microsoft C compiler:

```
cl /AH /Ot  device.c ngpib_io.c
```

Compiling with the Borland C compiler:

```
bcc -mh -O2   device.c ngpib_io.c
```

Compiling a C Device Program with the Interrupt ESP Package

In addition to any other necessary header file for the device program, you should include the following header files for the interrupt-driven package:

```
#include "io_param.h"
#include "gpib_io.h"
#include "inter_io.h"    (if you are using interrupt functions)
#include "queue_io.h     (if you are using queues)
```

You should include the following compiling options when you compile the device code:

```
/AH, -mh     for the Huge memory model
/Ot, -O2     to optimize for execution speed (optional)
/Gs          turn off stack checking
```

Compiling with the Microsoft C compiler:

`cl /AH /Ot /Gs device.c gpib_io.c inter_io.c` (`queue_io.c` if using queues)

Compiling with the Borland C compiler:

`bcc -mh -O2  device.c gpib_io.c inter_io.c` (`queue_io.c` if using queues)

GPIB Function Descriptions

The remainder of this chapter is a quick reference for TNT4882 ESP-488TL interrupt and noninterrupt package functions.

Status Functions

Use these functions to inquire about or set the GPIB status of the TNT4882.

Function	Description
`Clear_4882_Status(int reg, int mask)`	Clears a field of selected bits from an IEEE 488.2 status register, then updates the Serial Poll Byte (STB).
`Read_4882_Status(int reg)`	Returns an integer value from an IEEE 488.2 status register.
`Set_4882_Status(int reg, int mask)`	Sets a field of selected bits in an IEEE 488.2 status register, then updates the STB.
`Update_INTERFACE_STATUS()`	Updates the `INTERFACE_STATUS` status word.
`Wait_For_Interface(int mask)`	Waits until the mask event occurs in the `INTERFACE_STATUS` status word.

Initialization Functions

Use these functions to set up the operating modes of the TNT4882.

Function	Description
`High_Speed_Select(int hs_mode, int cable_length, int enable)`	Initializes the HS488 timing registers on the TNT4882.
`Initialize_Interface()`	Initializes the TNT4882 to an initial state.
`Interface_Off()`	Disables the TNT4882 from sending or receiving data and commands.
`Set_Timeout(int factor_index, int byte_timeouts)`	Sets the timeout time for TNT4882 I/O transfers.

Addressing Functions

Use these functions to set up the addressing modes of the TNT4882.

```
Change_Multiple_Addresses(int *address_list,
                          int num_of_addresses)
```
Changes the multiple address list to a new set of multiple primary/secondary addresses.

```
Change_Primary_Address(int address)
```
Changes the primary GPIB address to a new address.

```
Change_Secondary_Address(int address)
```
Changes the secondary GPIB address to a new address.

```
Set_Address_Mode(int mode)
```
Configures the TNT4882 addressing modes.

I/O Functions

Use these functions to read and write date either from or to the GPIB.

```
Abort_ASYNC_IO()
```
Removes I/O interrupts and resets the TNT4882 FIFOs.

```
Read_GPIB_Lines()
```
Reads the state of the GPIB hardware lines.

```
Receive(char *buf, unsigned long int cnt, int term)
```
Reads data from the bus.

```
Receive_ASYNC(char *buf, unsigned long int cnt,
              int term)
```
Sets up the read interrupts and returns immediately.

```
Send(char *buf, unsigned long int cnt, int term)
```
Writes data to the bus.

```
Send_ASYNC(char *buf, unsigned long int cnt, int term)
```
Sets up the write interrupts and returns immediately.

Interrupt Control Functions

Use these functions to set and clear user-interrupt functions.

`Clear_Interrupt_Function(int reg, int bit, void *function())`
Disables a function from occurring on a certain interrupt condition.

`Set_Interrupt_Function(int reg, int bit, void *function())`
Enables a function to be called on a certain interrupt condition.

Chapter 3
TNT4882 Talker/Listener ESP Functions and Utilities Reference

This chapter explains how to use the functions and utilities in the TNT4882 ESP-488TL package. This chapter lists the functions in alphabetical order.

The function descriptions in this chapter discuss the following information:

- Function Prototype — States the prototype of how the function is declared.
- Source File — States where the function is declared.
- Header File — States the header file for the function.
- Related Functions — List other associated functions.
- Purpose — Gives a short explanation of the function.
- Description — Gives a detailed description of the function.
- Parameters — List the valid parameters for the argument list.
- Equivalents — List equivalent functions in the source code.
- Examples — List examples of usage.

INTERRUPT ESP ONLY

void Abort_ASYNC_IO(void)

Source File: `gpib_io.h` (macro)
Header File: `gpib_io.h`
Related Functions: `Send_ASYNC()`, `Receive_ASYNC()`

Purpose

Terminate asynchronous I/O operation.

Description

`Abort_ASYNC_IO()` calls `DONE_Interrupt_Handler()` to remove interrupt functions and reset the FIFOs of the TNT4882.

If I/O is disabled before it can terminate normally, an Error Aborted (EABO) error is set. If EABO did not occur, the transfer actually completed before the `Abort_ASYNC_IO()` function could stop it.

Parameters

None.

Equivalents

```
Abort_ASYNC_IO() = IO_Control(ABORT);
```

Example

1. Stop an asynchronous I/O call.

```
Receive_ASYNC(buf, 1000, EOI);   Set up read.
Wait_For_Interface(UCMPL|TIMO|ERR|TACS|NACS);
                                 Wait for completion or an error.
if (INTERFACE_STATUS&(TACS|NACS))
                                 If address state changed, stop I/O.
   Abort_ASYNC_IO();
```

void Change_Primary_Address(int new_address)

Source File: `ngpib_io.c` or `gpib_io.c`
Header File: `ngpib_io.h` or `gpib_io.h`
Related Functions: `Change_Secondary_Address()`, `Set_Address_Mode()`

Purpose

Change the primary GPIB address to a new address.

Description

`Change_Primary_Address()` writes the new address to the TNT4882 address register, then saves the new address in a `Primary_Address` global variable that is used by `Initialize_Interface()`. If `Initialize_Interface()` is called after `Change_Primary_Address()`, the TNT4882 will be reinitialized with its new address. This function returns EARG if the TNT4882 is not configured for address modes 0, 1, or 2.

The `io_param.h` header file has a `PRIMARY_ADDRESS` entry for the initial address setting.

Parameters

`new_address` = 0 to 30	New primary address of device.

Equivalents

None.

Examples

1. Change the address.

`Set_Address_Mode(0);`	Set to single primary addressing.
`Change_Primary_Address(6);`	Set the TNT4882 to address 6.

2. Reinitialize the interface.

`io_param.h ... #define PRIMARY_ADDRESS 2`

`Initialize_Interface();`	Set up the TNT4882, GPIB address 2.
`Change_Primary_Address(5);`	Set the TNT4882 to address 5.
`Initialize_Interface();`	Reinitialize the TNT4882; set the address to 5.

void Change_Multiple_Addresses(int *address_list, int num_of_addresses)

Source File: `ngpib_io.c` or `gpib_io.c`
Header File: `ngpib_io.h` or `gpib_io.h`
Related Functions: `Set_Address_Mode()`

Purpose

Change the primary address list to a new set of primary addresses.

Description

`Change_Primary_Addresses()` updates the `Primary_Address_List`, which is a global buffer of possible primary addresses that the interface can recognize. Only the `Validate_Primary_Address()` function uses this list to verify a new primary address in the Command Pass Through Register (CPTR). The `io_param.h` header file has a `PRIMARY_ADDRESS_LIST` entry and a `USE_PRIMARY_ADDRESSES (YES/NO)` preprocessor constant for the initial address setting. This function returns EARG if the TNT4882 is not configured for address modes 2 or 3.

Parameters

`address_list` = {0, 1, 2, 3 ... 30}	Address list can be any combination of values from 0 to 30.
`num_of_addresses` = 1 to 30	Number of entries in list.

Equivalents

None.

Examples

1. Change the address list.

```
int new_address_list[4] = {2, 5, 6, 12};
Set_Address_Mode(3);          Set to multiple primary address mode.
Change_Multiple_Addresses(new_address_list, 4);
```

2. Reinitialize the interface.

```
io_param.h ...
#define ADDRESS_MODE 3
#define PRIMARY_ADDRESS_LIST {25, 26, 27}
#define NUMBER_OF_ADDRESSES 3

int new_address_list []= {2, 5, 6, 12}
Initialize_Interface();       Set primary addresses to 25, 26, and 27.
```

`Change_Multiple_Addresses(new_address_list, 4);`	
	Change primary addresses to 2, 5, 6, and 12.
`Initialize_Interface();`	Set primary addresses to 2, 5, 6, and 12 again.

void Change_Secondary_Address(int new_address)

Source File: `ngpib_io.c` or `gpib_io.c`
Header File: `ngpib_io.h` or `gpib_io.h`
Related Functions: `Change_Primary_Address()`, `Set_Address_Mode()`

Purpose

Change the secondary GPIB address to a new address.

Description

`Change_Secondary_Address()` writes a new secondary address to the TNT4882 address register and places the TNT4882 into secondary addressing mode. `Change_Secondary_Address()` then saves the new address in a `Secondary_Address` global variable that is used by `Initialize_Interface()`. If `Initialize_Interface()` is called at a later time, the interface will be reinitialized, but it will keep its new address. This function returns EARG if the TNT4882 is not configured for address mode 1.

The `io_param.h` header file has a `SECONDARY_ADDRESS` entry and a `USE_SECONDARY_ADDRESS (YES/NO)` preprocessor constant for the initial address setting.

Parameters

`new_address` = 0 to 30	New secondary address.

Equivalents

None.

Examples

1. Change the secondary address.

`Set_Address_Mode(1);`	Set to secondary addressing mode.
`Change_Primary_Address(1);`	Change primary address.
`Change_Secondary_Address(6);`	Change the secondary address to 6 or (MSA = 0x66).

2. Reinitialize the interface.

`io_param.h ...`	
`#define SECONDARY_ADDRESS 2`	
`Initialize_Interface();`	Set the secondary address to 2.
`Change_Secondary_Address(5);`	Change the secondary address to 5.
`Initialize_Interface();`	Set the secondary address to 5 again.

void Clear_4882_Status(int reg, int mask)

Source File: `ngpib_io.h` (macro) or `gpib_io.h` (macro)
Include File: `ngpib_io.h` or `gpib_io.h`
Related Functions: `Set_4882_Status()`, `Read_4882_Status()`

Purpose

Clear a field of bits from an IEEE 488.2 status register (reg).

Description

`Clear_4882_Status()` clears status bits that are defined by `mask` in the status register defined by `reg`, then updates the Serial Poll Byte (STB).

The IEEE 488.2 registers already defined by the `ngpib_io.h` or `gpib_io.h` files and required by the IEEE 488.2 standard are listed below under *Parameters*. You can easily add other status registers that are defined by the IEEE 488.2 standard to the `Update_4882_Status()` function and to the list of defined constants in the header files `ngpib_io.h` and `gpib_io.h.`

Parameters

488.2 registers

`reg` = STB (0)	Status byte.
SRE (1)	Service Request Enable.
ESE (2)	Event Status Byte (ESB) Enable.
ESR (3)	Event Status Register.
IST (4)	Individual Status bit (used for parallel polls).
`mask` = 0x00 to 0xFF	Bits to be cleared.

Equivalents

```
Clear_4882_Status() = Update_4882_Status(CLEAR,,,)
```

Examples

1. Clear the status bits.

`Clear_4882_Status(ESR, 0x01);`	Clear the Operation Complete status bit.
`Clear_4882_Status(SRE, 0xff);`	Disable all Service Request (SRQ) setting events.
`Clear_4882_Status(STB, 0x10);`	Clear the Message Available status bit.

2. Turn off the IST bit.

```
Clear_4882_Status(IST, 0x01);
```

INTERRUPT ESP ONLY

void Clear_Interrupt_Function(int reg, int mask, void (*function)())

Source File: `inter_io.h` (macro)
Header File: `inter_io.h`
Related Functions: `Set_Interrupt_Function()`

Purpose

Remove a function from the list of interrupt routines.

Description

`Clear_Interrupt_Function()` removes a function from a list of interrupt routines associated with a specific interrupt mask register (IMR0, IMR1, IMR2, or IMR3).

For the interrupt ESP, every interrupt mask register has a function pointer array associated with it. Every function pointer array has eight entries that correspond to each bit of the interrupt mask register. `Clear_Interrupt_Function()` removes this function pointer entry.

Parameters

`reg` = R_imr0, R_imr1, R_imr2, R_imr3	TNT4882 mask registers.
`bit` = (1<<0), (1<<1), (1<<2) . . . (1<<7)	Bit in the mask the function is associated with.

`function` = user-defined function (for example: `User_Function()`)

Equivalents

```
Clear_Interrupt_Function() =
Interrupt_Function_Control(,,,CLEAR)
```

Example

1. Set an interrupt function on Device Clear (DEC).

`void main(void)`	Define the main routine.
`{`	
`    Initialize_Interface();`	Initialize the TNT4882.
`    Set_Interrupt_Function(R_imr1,B_dec,Device_Clear);`	Set interrupt on the Device Clear (DEC) bit.
`    ..........`	
`}`	
`void Device_Clear(void)`	Define the interrupt function.
`{`	
`    TNT_Out(R_auxmr, B_clrDEC)`	On interrupt, clear the DEC bit (B_dec). Turn off further DEC interrupts.

```
Clear_Interrupt_Function(R_imr1, B_dec,   Device_Clear);
    ..........
                                       };
```

void High_Speed_Select(int cable_length, int enable)

Source File: `ngpib_io.c` or `gpib_io.c`
Include File: `ngpib_io.h` or `gpib_io.h`
Related Functions: `Initialize_Interface()`

Purpose

Initialize the HS488 timing registers and place the TNT4882 into *one-chip* mode.

Description

`High_Speed_Select()` writes to the TNT4882's HS488 timing register values that are based on the total GPIB cable length. The maximum handshake speed is 8 MHz for 2 m. The slowest handshake speed is 1.5 MHz for 15 m. The TNT4882 automatically detects whether high-speed transfers can be undertaken between the Talker and Listener(s). `Initialize_Interface()` calls `High_Speed_Select()` to place the TNT4882 into one-chip mode. See the *Configuring the TNT4882 Handshake Mode and High-Speed I/O* section in Chapter 6, *Advanced Topics*.

Parameters

`cable_length` = 0 to 15	Total length of cable in meters.
`enable` = ENABLE (1), DISABLE (0)	Turn on/off high-speed mode.

Equivalents

None.

Examples

1. Use the HS488 protocol with 5 m of cable.
```
High_Speed_Select(5, ENABLE);
```

2. Turn off the HS488 protocol and use the three-wire GPIB handshake. (`x` represents don't cares.)
```
High_Speed_Select(x,DISABLE);
```

void Initialize_Interface()

Source File: `ngpib_io.c` or `gpib_io.c`
Include File: `ngpib_io.h` or `gpib_io.h`
Related Functions: `Interface_Off()`

Purpose

Initialize the TNT4882 to be an IEEE 488.2 GPIB interface.

Description

`Initialize_Interface()` should be the first GPIB function called in any device program. You can use `Initialize_Interface()` to reinitialize the interface to its default state.

`Initialize_Interface()` sets the interface into a power-on default state, configures the addressing mode and the high-speed-handshaking modes, and writes to several auxiliary command registers. For the interrupt ESP, it loads interrupt vectors to handle the I/O queues and multiple secondary addressing. You can also use `Initialize_Interface()` to set interrupts for Device Clear and Device Trigger events. TNT4882 initialization is explained in greater detail in the *TNT4882 Initialization* section in Chapter 6, *Advanced Topics*.

Parameters

None.

Equivalents

None.

Example

1. Initialize the interface.
   ```
   Initialize_Interface();
   ```

void Interface_Off()

Source File: `ngpib_io.h` (macro) or `gpib_io.h` (macro)
Header File: `ngpib_io.h` or `gpib_io.h`
Related Functions: `Initialize_Interface()`

Purpose

Disable the TNT4882 from sending or receiving data or commands.

Description

`Interface_Off()` issues a chip reset command to the AUXMR, placing the handshaking functions into their idle states and disabling the primary address in the address register. For the interrupt ESP, this function also unloads and disables TNT4882 interrupts.

`Interface_Off()` should be the last GPIB command issued in a device program.

Parameters

None.

Equivalents

```
Interface_Off() = IO_Control(DISABLE)
```

Examples

1. Turn off and on I/O capability.

`Initialize_Interface();`	Initialize the TNT4882.
`..........`	
`Interface_Off();`	Disable TNT4882 data reception and data transmission.
`..........`	
`Initialize_Interface();`	Enable TNT4882 data reception and data transmission.

2. Turn off I/O.

`void main(void)`	
`{`	
`   Initialize_Interface();`	Initialize the TNT4882.
`   ..........`	
`   ..........`	
`   Interface_Off();`	Turn off the TNT4882 (if using the interrupt ESP, this function unloads interrupts).
`}`	

int Read_4882_Status(int reg)

Source File: `ngpib_io.h` (macro) or `gpib_io.h` (macro)
Include File: `ngpib_io.h` or `gpib_io.h`
Related Functions: `Set_4882_Status()`, `Clear_4882_Status()`

Purpose

Return a value from an IEEE 488.2 status register (reg).

Description

`Read_4882_Status()` directly reads the IEEE 488.2 status memory register value.

Parameters

488.2 registers

`reg` = STB (0)	Status byte.
SRE (1)	Service Request Enable.
ESE (2)	Event Status Byte (ESB) Enable.
ESR (3)	Event Status Register.
IST (4)	Individual Status bit (used for parallel polls).

Equivalents

None.

Example

1. Read the STB and ESR memory registers.

```
serial_poll_byte = Read_4882_Status(STB);
```
Read the STB byte.
```
event_status_reg = Read_4882_Status(ESR);
```
Read the ESR byte.

int Read_GPIB_Lines()

Source File: `ngpib_io.h` (macro)
Header File: `ngpib_io.h`
Related Functions: `Receive_ASYNC()`, `Send()`, `Send_ASYNC()`, `Receive()`

Purpose

Read the state of the GPIB hardware lines.

Description

`Read_GPIB_Lines()` returns the current state of the GPIB hardware lines. The upper byte contains the state of the handshake and control lines; the lower byte contains the data line state of the bus.

Format of word returned:

upper byte	ATN, DAV, NDAC, NRFD, EOI, SRQ, IFC, REN
lower byte	DIO8, DIO7, DIO6, DIO5, DIO4, DIO3, DIO2, DIO1

Parameters

None.

Equivalents

None.

Examples

1. Read the data lines.
```
data_status = Read_GPIB_Lines() & 0x0f;
```

2. Read the interface management lines.
```
management_line_status = Read_GPIB_Lines() & 0xf0;
```

void Receive(char *buf,unsigned long int cnt,int term)

Source File:	`ngpib_io.h` (macro) or `gpib_io.h` (macro)
Header File:	`ngpib_io.h` or `gpib_io.c`
Related Functions:	`Receive_ASYNC()`, `Send()`, `Send_ASYNC()`, `Read_GPIB_Lines();`

Purpose

Read data from the GPIB into a buffer.

Description

`Receive()` initially clears I/O status bits and I/O count global `DATA_COUNT`, then uses either DMA or programmed I/O to read data from the GPIB. The selection of DMA or programmed I/O depends on the `io_param.h` configuration file option `USE_DMA`. If `USE_DMA = YES`, DMA is used to transfer the data to the buffer. Programmed I/O is used to transfer data to the buffer only if DMA is not used. The type of termination that is detected in the transfer depends on the parameter `term`.

After the function terminates, the number of data bytes actually transferred over the GPIB is contained in the `DATA_COUNT` global variable and the `INTERFACE_STATUS` variable is updated. If the device program uses multiple addresses, `MULTI_ADDRESS` contains the address that the data was received at.

A more detailed description of `Receive()` is in the *GPIB I/O with the TNT4882* section in Chapter 6, *Advanced Topics*.

Parameters

`buf` = pointer to data location

`cnt` = 0 to 0xffffffff (4 GB)	Number of bytes to read from the GPIB.
`term` = EOI (0x2000)	Terminate when the TNT4882 receives EOI.
= EOS (0x1000)	Terminate when the TNT4882 receives EOS.

Equivalents

1. If the `io_param.h` configuration file `USE_DMA = YES`, then
   `Receive() = User_GPIB_IO(INPUT, DMA,,,SYNC);`

2. If the `io_param.h` configuration file `USE_DMA = NO`, then
   `Receive() = User_GPIB_IO(INPUT, PROG,,,SYNC);`

Examples

1. Read data from the bus.

```
Wait_For(LACS);                  Wait to be listen addressed.
Receive(buf, 1000, EOI);         Reads up to 1000 bytes of data
                                 from the bus.
```

2. Read data from the bus if DMA is enabled.

```
Wait_For(LACS);                  Wait to be listen addressed.
User_GPIB_IO(INPUT, DMA, buf, 1000, EOI, SYNC);
                                 Read up to 1000 bytes of data
                                 from the bus.
```

3. Terminate on the EOS byte.

```
Update_INTERFACE_STATUS():       Update the status word.
if (INTERFACE_STATUS&LACS){      Check for listen address.
   Receive(buf, 13, EOI|EOS);
                                 Read up to 13 bytes into the buffer
                                 and terminate on EOI or EOS.
}
```

4. Use timeouts with I/O calls.

```
Set_Timeout(13,ENABLE);          Enable timeouts 17 s after
                                 each byte.
.........
User_GPIB_IO(INPUT, PROG, buf, 0xffff, NONE, SYNC);
                                 Read 64 KB from the bus by using
                                 programmed I/O.
```

5. Use multiple primary addresses (3, 8, 15).

```
Receive(buf, 1000, EOI|EOS);     Read 1000 bytes.
switch(MULTI_ADDRESS){           Select what to do.
   case(0x0300):                 Address 3.
      ..........
      break;
   case(0x0800):                 Address 8.
      ..........
      break;
   case(0x0F00):                 Address 15.
      ..........
      break;
}
```

INTERRUPT ESP ONLY

void Receive_ASYNC(char *buf,unsigned long int cnt, int term)

Source File: `gpib_io.h` (macro)
Header File: `gpib_io.h`
Related Functions: `Receive()`, `Send()`, `Send_ASYNC()`, `Read_GPIB_Lines();`

Purpose

Input or read data from the GPIB into the buffer. Return immediately after interrupts are set up to input data.

Description

`Receive_ASYNC()` initially clears I/O status bits and I/O count global `DATA_COUNT`, then uses either DMA or programmed I/O to read data from the GPIB. The selection of DMA or programmed I/O depends on the `io_param.h` configuration file option `USE_DMA`. If `USE_DMA = YES`, DMA is used to transfer the data to the buffer. Programmed I/O is used to transfer data to the buffer only if DMA is not used. The type of termination that is detected depends on the `term` parameter. The User Complete (UCMPL) bit can be used to detect when the transfer has completed.

After the function terminates, the number of data bytes actually transferred over the GPIB is contained in the `DATA_COUNT` global variable and the `INTERFACE_STATUS` variable is updated. If the device program uses multiple addresses, `MULTI_ADDRESS` contains the address that the data was received at.

A more detailed description of `Receive_ASYNC()` is in the *GPIB I/O with the TNT4882* section in Chapter 6, *Advanced Topics*.

Parameters

`buf` = pointer to data location

`cnt` = 0 to 0xffffffff (4 GB)	Number of bytes to read from the GPIB.
`term` = EOI (0x2000)	Terminate when the TNT4882 receives EOI.
= EOS (0x1000)	Terminate when the TNT4882 receives EOS.

Equivalents

1. If the `io_param.h` configuration file `USE_DMA = YES`, then
   `Receive_ASYNC() = User_GPIB_IO(INPUT, DMA,,, ASYNC);`

2. If the `io_param.h` configuration file `USE_DMA = NO`, then
   `Receive_ASYNC() = User_GPIB_IO(INPUT, PROG,,, ASYNC);`

Examples

1. Read data from the bus.

```
Wait_For(LACS);                      Wait to be listen addressed.
Receive_ASYNC(buf, 1000, EOI);
                                     Read up to 1000 bytes of data
                                     from the bus asynchronously.
```

2. Read data from the bus if DMA is enabled.

```
Wait_For(LACS);                      Wait to be listen addressed.
User_GPIB_IO(INPUT, DMA, buf, 1000, EOI, ASYNC);
                                     Read up to 1000 bytes of data
                                     from the bus asynchronously.
```

3. Terminate on the EOS byte.

```
Update_INTERFACE_STATUS():           Update the status word.
if (INTERFACE_STATUS&LACS){          Check for listen address.
   Receive_ASYNC(buf, 13, EOI|EOS);
                                     Read up to 13 bytes into the buffer
                                     and terminate on EOI or EOS.
}
```

4. Use timeouts.

```
Set_Timeout(13, ENABLE)              Enable 17-s timeouts after
                                     each byte.
..........
User_GPIB_IO(INPUT, PROG, buf, 0xffff, NONE, ASYNC);
                                     Read 64 KB from the bus by using
                                     programmed I/O.
```

5. Use multiple primary addresses (3, 8, 15).

```
Receive_ASYNC(buf, 1000, EOI/EOS);
                                     Read 1000 bytes.
Wait_For_Interface(UCMPL);           Wait for completion.
switch(MULTI_ADDRESS){               Select what to do.
   case(0x0300):                     Address 3.
      ..........
      break;
   case(0x0800):                     Address 8.
      ..........
      break;
   case(0x0F00):                     Address 15.
      ..........
      break;
}
```

void Send(char_buf *buf, unsigned long cnt, int term)

Source File: `ngpib_io.h` (macro), `gpib_io.h` (macro)
Header File: `ngpib_io.h`, `gpib_io.h`
Related Functions: `Receive()`, `Send_ASYNC()`, `Receive_ASYNC()`, `Read_GPIB_Lines()`

Purpose

Write data in the buffer to the GPIB.

Description

`Send()` initially clears I/O status bits and the I/O count global `DATA_COUNT`, then uses either DMA or programmed I/O to write data to the GPIB. The selection of DMA or programmed I/O depends on the `io_param.h` configuration file option `USE_DMA`. If `USE_DMA = YES`, DMA is used to transfer the data to the buffer. Programmed I/O is used to transfer data to the buffer only if DMA is not used. Parameter `term` sets the type of termination that is used.

After the function terminates, the number of data bytes actually transferred over the GPIB is contained in the `DATA_COUNT` global variable and `INTERFACE_STATUS` is updated. If the device program uses multiple addressing, `MULTI_ADDRESS` can be set to the address that the TNT4882 will write data from.

A more detailed description of `Send()` is in the *GPIB I/O with the TNT4882* section in Chapter 6, *Advanced Topics*.

Parameters

`buf` = pointer to data location

`cnt` = 0 to 0xffffffff (4 GB)	Number of bytes to write to the GPIB.
`term` = EOI (0x2000)	Set the EOI line when transmitting the last data byte.
= EOS (0x1000)	Set the EOI line when EOS is received to be output.
= NONE (0)	Do nothing at the end of the transfer.

Equivalents

1. If the `io_param.h` configuration file `USE_DMA = YES`, then
   `Send() = User_GPIB_IO(OUTPUT, DMA,,,SYNC);`

2. If the `io_param.h` configuration file `USE_DMA = NO`, then
   `Send() = User_GPIB_IO(OUTPUT, PROG,,,SYNC);`

 © National Instruments Corp.

Examples

1. Write data to the bus.

```
Wait_For(TACS);                   Wait to be talk addressed.
Send ("Hello World," 11, EOI);    Write Hello World to the bus
                                  and terminate with EOI.
```

2. Write data to the bus if DMA is enabled.

```
Wait_For(TACS);                   Wait to be talk addressed.
User_GPIB_IO(OUTPUT, DMA, "Hello World," 11, EOI,
SYNC);                            Write Hello World to the bus
                                  and terminate with EOI.
```

3. Write data to the bus and assert EOI with the EOS byte.

```
Update_INTERFACE_STATUS();        Update the status word.
if (INTERFACE_STATUS&TACS){       Check for talk address state.
   Send("Hello World \n," 13, EOI | EOS);
                                  Write data to the bus.
}
```

4 Use timeouts with I/O.

```
Set_Timeout(13, ENABLE)           Set the timer to timeout 17 s after
                                  each byte.
..........
User_GPIB_IO(OUTPUT, PROG, buf, 0xffff, NONE, SYNC);
                                  Write 64 KB from the buffer to
                                  the GPIB.
```

5. Write from a specific multiple address (3, 8, 15).

```
Switch(CURRENT_ADDRESS){
   case(0x0300):                  Write from address 3.
      MULTI_ADDRESS = 0x0300;
      Send("I'M ADDRESS 3," 12, EOI);
      break;
   case(0x0800):                  Write from address 8.
      MULTI_ADDRESS = 0x0800;
      Send("I'M ADDRESS 8," 12, EOI);
      break;
   case(0x0F00):                  Write from address 15.
      ..........
      break;
}
```

INTERRUPT ESP ONLY

void Send_ASYNC(char_buf *buf, unsigned long cnt, int term)

Source File:	`gpib_io.h` (macro)
Header File:	`gpib_io.h`
Related Functions:	`Receive()`, `User_GPIB_IO()`, `Send()`, `Receive_ASYNC()`, `Read_GPIB_Lines()`

Purpose

Write data in the buffer to the GPIB and return immediately after interrupts are set up to output data.

Description

`Send_ASYNC()` initially clears I/O status bits and I/O count global `DATA_COUNT`, then uses either DMA or programmed I/O to write data to the GPIB. The selection of DMA or programmed I/O depends on the `io_param.h` configuration file option `USE_DMA`. If `USE_DMA = YES`, DMA is used to transfer the data to the buffer. Programmed I/O is used to transfer data to the buffer only if DMA is not used. The parameter `term` sets the type of termination that is used. You can use the UCMPL bit to detect when the transfer has completed.

After the function terminates, the number of data bytes actually transferred over the GPIB is contained in the `DATA_COUNT` global variable and `INTERFACE_STATUS` is updated. If the device program uses multiple addressing, `MULTI_ADDRESS` can be set to the address the TNT4882 will write data from.

A more detailed description of `Send_ASYNC()` is in the *GPIB I/O with the TNT4882* section in Chapter 6, *Advanced Topics*.

Parameters

`buf` = pointer to data location

`cnt` = 0 to 0xffffffff (4 GB)	Number of bytes to write to the bus.
`term` = EOI (0x2000)	Set the EOI line when transmitting the last data byte.
= EOS (0x1000)	Set the EOI line when EOS is received to be output.
= NONE (0)	Do nothing at the end of the transfer.

Equivalents

1. If the `io_param.h` configuration file `USE_DMA = YES`, then
   `Send_ASYNC() = User_GPIB_IO(OUTPUT, DMA,,,ASYNC);`

2. If the `io_param.h` configuration file `USE_DMA = NO`, then
   `Send_ASYNC() = User_GPIB_IO(OUTPUT, PROG,,,ASYNC);`

Examples

1. Write data to the bus asynchronously.

```
Wait_For(TACS);                      Wait to be talk addressed.
Send_ASYNC("Hello World," 11, EOI);
                                     Write Hello World to the bus
                                     and terminate with EOI.
```

2. Write data to the bus if DMA is enabled.

```
Wait_For(TACS);                      Wait to be talk addressed.
User_GPIB_IO(OUTPUT, DMA, "Hello World," 11, EOI,
ASYNC);                              Write Hello World to the bus
                                     and terminate with EOI.
```

3. Write data to the bus and assert EOI with the EOS byte.

```
Update_INTERFACE_STATUS();           Update the status word.
if (INTERFACE_STATUS&TACS){          Check for talk address state.
   Send_ASYNC("Hello World \n," 13, EOI|EOS);
                                     Write data to the bus.
}
```

4. Use timeouts with I/O.

```
Set_Timeout(13, ENABLE)              Set the timer to timeout 17 s
                                     after each byte.
..........
User_GPIB_IO(OUTPUT, PROG, buf, 0xffff, NONE,
ASYNC);                              Write 64 KB from the buffer to
the GPIB.
```

5. Write from a specific multiple address (3, 8, 15).

```
Switch(CURRENT_ADDRESS){
   case(0x0300):                     Write from address 3.
      MULTI_ADDRESS = 0x0300;
      Send_ASYNC()("I'M ADDRESS 3," 12, EOI);
      break;
   case(0x0800):                     Write from address 8.
      MULTI_ADDRESS = 0x0800;
      Send_ASYNC()("I'M ADDRESS 8," 12, EOI);
      break;
   case(0x0F00):                     Write from address 15.
      ..........
      break;
}
```

void Set_4882_Status(int reg, int mask)

Source File: `ngpib_io.h` (macro) or `gpib_io.h` (macro)
Include File: `ngpib_io.h` or `gpib_io.h`
Related Functions: `Clear_4882_Status()`, `Read_4882_Status()`

Purpose

Set a field of bits in an IEEE 488.2 status register (reg).

Description

`Set_4882_Status()` writes the bits in the mask to the memory register defined by `reg`, then updates the STB (the 488.2 Status Byte).

The IEEE 488.2 registers already defined by the `ngpib_io.h` or `gpib_io.h` files and required by the IEEE 488.2 standard are listed below under *Parameters*. You can easily add other status registers defined by the IEEE 488.2 standard to the `Update_4882_Status()` function and to the list of defined constants in the header file `ngpib_io.h` or `gpib_io.h`. For more information, see the *IEEE 488.2 Status Model* section in Chapter 6, *Advance Topics*.

Parameters

488.2 registers

`reg` = STB (0)	Status byte.
SRE (1)	Service Request Enable.
ESE (2)	Event Status Byte (ESB) Enable.
ESR (3)	Event Status Register.
IST (4)	Individual Status bit (used for parallel polls).

mask = 0x00 to 0xFF

Equivalents

```
Set_4882_Status() = Update_4882_Status(SET,,,)
```

Examples

1. Set the SRQ signal on ESB (refer to the IEEE 488.2 standard).

`Set_4882_Status(ESE, 0x01);`	Enable the setting of the ESB bit in STB upon Operation Complete.
`Set_4882_Status(SRE, 0x20);`	Enable SRQ on ESB.
`..........`	
`Set_4882_Status(ESR, 0x11);`	Set the ESE Execution Error and the Operation Complete status bits. This action causes an SRQ to occur.

2. Set the IST bit.

`Set_4882_Status(IST, 0x01);`	Turn on the IST bit for parallel polls.

void Set_Address_Mode(int mode)

Source File: `ngpib_io.c` or `gpib_io.c`
Header File: `ngpib_io.h` or `gpib_io.h`
Related Functions: `Change_Primary_Address()`, `Change_Secondary_Address()`, `Change_Multiple_Addresses()`

Purpose

Change the addressing mode.

Description

`Set_Address_Mode()` writes to the address mode register of the TNT4882 and configures the TNT4882 to one of seven address modes. `Set_Address_Mode()` must be followed by one or more of the related functions in order to configure the address(es) the TNT4882 will recognize.

Parameters

`mode` =	
0	Single primary addressing mode.
1	Single primary–single secondary addressing mode.
2	Single primary–multiple secondary addressing mode.
3	Multiple primary addressing mode.
4	Talk-only mode.
5	Listen-only mode.
6	No-address mode.

Equivalents

None.

Examples

1. Set to multiple secondary addressing mode.

```
int ad_list[5]={1,2,3,4,5};          Set new address list;

Set_Address_Mode(2);                 Set to multiple secondary mode.
Change_Primary_Address(1);           Set primary address.
Change_Multiple_Addresses(ad_list5);
                                     Set secondary addresses.
```

2. Set to normal single primary addressing mode.

```
Set_Address_Mode(0);                 Set to primary address mode.
Change_Primary_Address(5);           Set primary address.
```

3. Set to multiple primary addressing mode.

```
int ad_list[5]={1,2,3,4,5};          Set new address list;
Set_Address_Mode(3);                 Set to multiple primary mode.
Change_Multiple_Addresses(ad_list 5);
                                     Set primary addresses.
```

4. Set to single primary–single secondary addressing mode.

`Set_Address_Mode(1);`	Set single address mode.
`Change_Primary_Address(1);`	Set primary address.
`Change_Secondary_Address(2);`	Set secondary address.

5. Set to listen-only mode.

```
Set_Address_Mode(5);
```

INTERRUPT ESP ONLY

void Set_Interrupt_Function(int reg, int mask, void (*function)())

Source File: `inter_io.h` (macro)
Header File: `nter_io.h`
Related Functions: `Clear_Interrupt_Function()`

Purpose

Insert a new interrupt function in the list of executable interrupt functions.

Description

`Set_Interrupt_Function()` enables an interrupt function by inserting the function pointer from the argument list into an array of functions associated with the register written in `reg`.

For the interrupt ESP, every interrupt mask register (IMR0, IMR1, IMR2, and IMR3) has an array of function pointers. Every array has eight entries corresponding to each bit in the interrupt mask. `Set_Interrupt_Function()` inserts the function pointer into the array at the index of the bit location in the mask.

For more information about interrupts, see the *Interrupts and the TNT4882 ESP-488TL* section in Chapter 6, *Advanced Topics*.

Parameters

`reg` = R_imr0, R_imr1, R_imr2, R_imr3	TNT4882 interrupt mask registers.
`bit` = (1<<0), (1<<1), (1<<2), ..(1<<7)	TNT4882 interrupt mask bits.
`function` = user-defined function	

Equivalents

```
Set_Interrupt_Function() =
Interrupt_Function_Control(,,,SET)
```

Example

1. Set interrupt on Device Clear (DEC).

```
void main(void)                           Define the main routine.
{
   Initialize_Interface();                Initialize the TNT4882.
   Set_Interrupt_Function(R_imr1, B_dec,
   Device_Clear);
                                          Set the interrupt on DEC.
..........
}
void Device_Clear(void)                   Define the interrupt function.
{
   TNT_Out(R_auxmr, B_clrDEC)             Clear the DEC bit on interrupt.
   Clear_Interrupt_Function(R_imr1,B_dec,Device_Clear);
                                          Disable Device_Clear
                                          function.
   ..........
};
```

void Set_Timeout(int factor_index, int byte_timeouts)

Source File: `ngpib_io.c` or `gpib_io.c`
Header File: `ngpib_io.h` or `gpib_io.h`
Related Functions: `Initialize_Interface()`

Purpose

Set timeout time for TNT4882 I/O transfers.

Description

`Set_Timeout()` writes to a global variable structure called `Timeout`. The `Timeout` structure contains the two values passed by this function. The `factor_index` is the value that must be written to Auxiliary Register J (AUXRJ) to set the correct timeout period. The actual waiting time is given by the `factor_index` column in Table 3-1. The parameter `byte_timeouts` is a flag that indicates whether the timer value represents the maximum time available to transfer a single byte or an entire data transfer. A `factor_index` of zero turns off any timeouts.

Table 3.1 Waiting Time

Factor Index	Seconds
0	off
1	16 μs
2	32 μs
3	128 μs
4	256 μs
5	1 ms
6	4 ms
7	16 ms
8	33 ms
9	131 ms
10	262 ms
11	1 s
12	4 s
13	17 s
14	34 s
15	134 s

Parameters

`factor_index`	= 0 to 15	Time to wait based on Table 3-1.
`byte_timeout`	= ENABLE (1)	Reset the timer to timeout on each byte.
	= DISABLE (0)	Timer holds the total time for the transfer.

Examples

1. Wait at least 4 ms for each byte to be sent or received.
   `Set_Timeout(6, ENABLE)`

2. Allow 1 s for entire transfer.
   `Set_Timeout(11, DISABLE)`

3. Disable timeouts. (`x` represents don't care.)
   `Set_Timeout(0,x);`

int Update_INTERFACE_STATUS()

Source File: `ngpib_io.c` or `gpib_io.c`
Include File: `ngpib_io.h` or `gpib_io.h`
Related Functions: `Wait_For_Interface()`

Purpose

Update the `INTERFACE_STATUS` status word.

Description

`Update_Interface_STATUS()` reads several interrupt status registers that are necessary for updating the status word. The `INTERFACE_STATUS` status word contains useful status bits that can inform the device program about the status of the TNT4882 and whether a GPIB event has occurred. For the noninterrupt ESP, this function also validates multiple addresses and the high-speed configuration command.

Table 3-2. INTERFACE_STATUS Status Word

Mnemonic	Bit Position	Description
ERR	15	GPIB Error
TIMO	14	Time Limit Exceeded
END	13	End Detected
EOS	12	End-of-String Detected
RQS	11	Requesting Service
IFC	10	Interface Clear
SPOLL	9	Serial Poll Active
UCMPL	8	User I/O Complete
LOK	7	Local Lockout
REM	6	Remote Programming
ASYNC	5	Asynchronous I/O In Progress
DTAS	4	Device Trigger Active State
DCAS	3	Device Clear Active State
LACS	2	Listener Active State
TACS	1	Talker Active State
NACS	0	Not Active State

Parameters

None.

Equivalents

None.

Examples

1. Get the current status into variable status.

```
status = Update_INTERFACE_STATUS();
```
Update the status word `INTERFACE_STATUS`.

2. Update the status to find out the address state.

Code	Comment
`Update_INTERFACE_STATUS();`	Update the status word `INTERFACE_STATUS`.
`if (INTERFACE_STATUS & LACS){`	Check for address state.
`   Receive(.....);`	Read data from the bus.
`   ..........`	
`}`	

void Wait_For_Interface(int mask)

Source File: `ngpib_io.c` or `gpib_io.c`
Include File: `ngpib_io.h` or `gpib_io.h`
Related Functions: `Update_INTERFACE_STATUS()`

Purpose

Wait until the mask event occurs in the `INTERFACE_STATUS` status word.

Description

`Wait_For_Interface()` calls `Update_INTERFACE_STATUS()` and compares the mask to the `INTERFACE_STATUS` word. When any condition in the mask is true, the call exits and the program continues. The mask variable relates to the status bits listed in Table 3-2.

Parameters

`mask` = 0x00 to 0xff	Mask related to the `INTERFACE_STATUS` bits.

Equivalents

None.

Examples

1. Update `INTERFACE_STATUS` and return.
   `Wait_For_Interface(0);`

2. Wait to be addressed.
   `Wait_For_Interface(LACS|TACS);`

3. Wait to be serial polled if `USE_SPOLL_BILL` = `YES`.
   `Wait_For_Interface(SPOLL);`

INTERRUPT ESP ONLY

Chapter 4
Queues

This chapter discusses the TNT4882 ESP-488TL I/O queues.

The TNT4882 interrupt ESP includes an interrupt handler and I/O functions to handle asynchronous circular data and event queues. The IEEE 488.2 standard calls these queues the Output Queue and the Input Buffer. In this manual, both are referred to as queues. These data queues are the gateway between the device program and the GPIB bus. All data written *to* the device passes into the Input Queue. All data written *from* the device passes into the Output Queue.

Queue Models

The TNT4882 ESP includes functions to individually enable or disable the queues. There are six I/O models (see Figure 4-1). Input queues are always on the left and output queues are always on the right. The block labeled *user* in Figure 4-1 is the device program that accesses the I/O queues. The block labeled *io* represents low-level calls that set up the TNT4882 and transmit data to the GPIB: this arrangement makes the addressing and I/O setup transparent to the *user* program. The figures lacking a queue but having an arrow that passes directly from *user* to *io* represent the Send/Receive functions that are generally used instead of a queue.

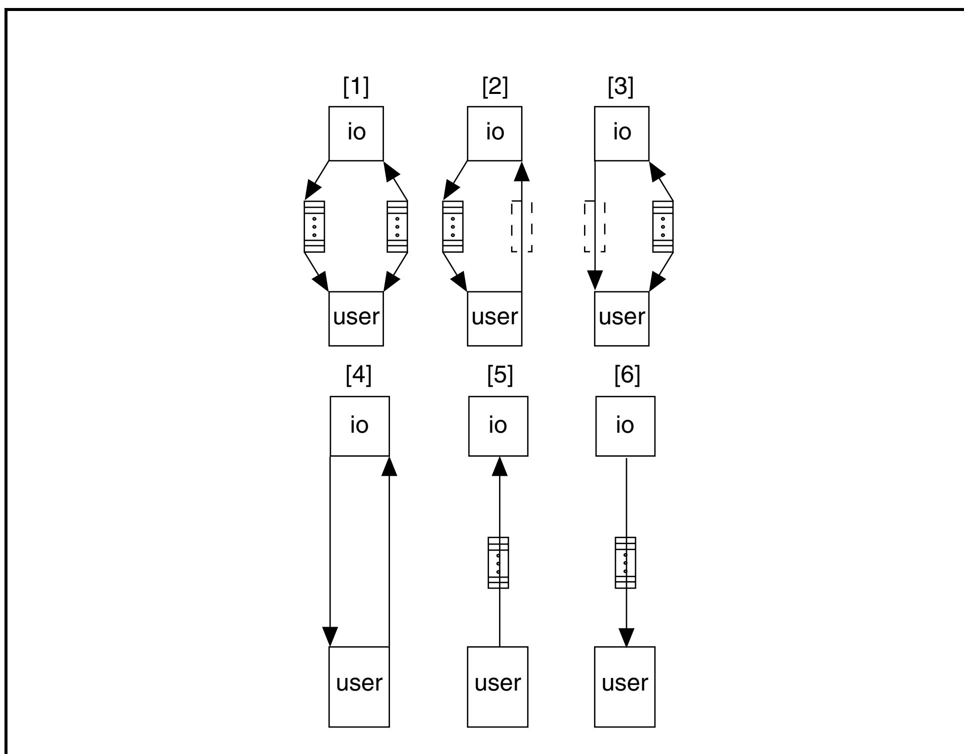


Figure 4-1. Queue Models

Six I/O Models

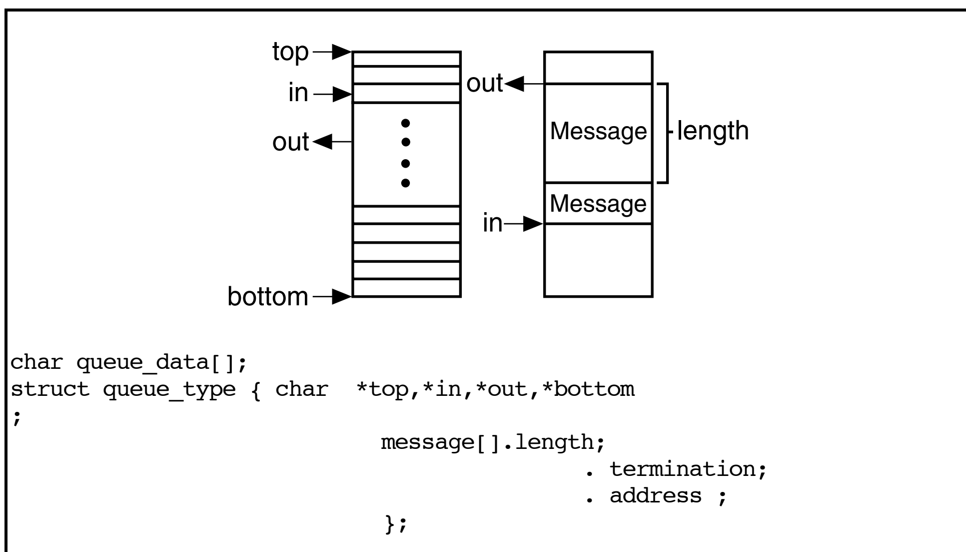
1. Both queues are enabled.
2. Output Queue is disabled; output handled by user send functions.
3. Input Queue is disabled; input handled by user receive functions.
4. Both queues are disabled; I/O handled by user.

5. Talk Only; I/O handled by user or queue.
6. Listen Only; I/O handled by user or queue.

Structure of a Queue

A queue consists of a contiguous data space; its size is set by the parameter `INPUT_QUEUE_SIZE` or `OUTPUT_QUEUE_SIZE` in the `io_param.h` configuration file. The Input Queue can hold multiple terminated messages, so the device program can receive more data through the asynchronous queue while it operates on some previous data message. In a similar manner, the Output Queue can have multiple messages, so the device can place more messages into the Output Queue while the `Queue_Interrupt_Handler()` writes data to the bus. The number of terminated messages that can be in the Output or Input Queue is set by `MAXIMUM_BUFFER_SIZE` in the `io_param.h` configuration file.

The pseudo C code in Figure 4-2 shows the queue structure. Two pointers represent the top and bottom of the queue data space and two pointers represent the place to insert new data *in* and remove data *out*. By using these four pointers, the `Queue_Interrupt_Handler()`, the `Read_Input_Queue()`, and the `Write_Output_Queue()` can access the data in the queues. The *message* entry is an array of information that describes the data messages located in the queue. The size of this message array structure is set by the `io_param.h` configuration file entry `MAXIMUM_BUFFER_SIZE`. The message structure keeps track of the message length, how the message is terminated, and what address the message is sent to when multiple addressing is used.




```
char queue_data[];
struct queue_type { char  *top,*in,*out,*bottom
;
                            message[].length;
                                           . termination;
                                           . address ;
                            };
```

Figure 4-2. Queue Structure

For events such as Device Trigger (DET) and Device Clear (DEC), the message entry is made in the termination field, not the data field. A pair of interrupt routines, `DET_Interrupt_Handler()` and `DEC_Interrupt_Handler()`, is included in the `gpib_io.c` file. These handlers insert the DTAS and DCAS events into the Input Queue. These routines *must* be modified for your particular device and they can be enabled with the `io_param.h` file constant `INCLUDE_EVENT_HANDLERS`. For more information, see the *Event Handling* section in Chapter 6, *Advanced Topics*.

Queue Information Globals

Three global variables convey important information about the queues.

`QUEUE_COUNT` Indicates the actual number of bytes read from or written to the data queues by the `Read_Input_Queue()` and `Write_Output_Queue()` functions.

`QUEUE_STATUS` Indicates the current state of both TNT4882 queues.

`QUEUE_ADDRESS` Indicates what address the TNT4882 was addressed as when it read the message—only if using multiple addressing.

QUEUE_STATUS

The `QUEUE_STATUS` status word reflects the current state of the asynchronous I/O queues. Table 4-1 lists the status bits and the bit positions in `QUEUE_STATUS`.

Table 4-1. QUEUE_STATUS Status Bits and Bit Positions

Mnemonic	Bit Position	Description
IQF	15	Input Queue Full
IQE	14	Input Queue Empty
IMAV	13	Input Message/Event Available
IDAV	12	Input Data Bytes Available
IMBF	11	Input Message Buffer Full
ISAV	10	Input Space Available
IQAC	9	Input Queue Active
IQEN	8	Input Queue Enabled
OQF	7	Output Queue Full
OQE	6	Output Queue Empty
OMWA	5	Output Message Waiting
ODWA	4	Output Data Bytes Waiting
OMBF	3	Output Message Buffer Full
OSAV	2	Output Space Available
OQAC	1	Output Queue Active
OQEN	0	Output Queue Enabled

IQF — The IQF bit indicates that all data space in the Input Queue is used. No more data will be received by the Input Queue while IQF is set.

IQE — The IQE bit indicates that no data is left in the Input Queue. Subsequent reads will return an Error Input Queue (EIQ) error because no data is available. Avoid an EIQ error by using the function `Message_Available()` or `Data_Available()` to determine if a terminated message or any data remains in the Input Queue.

IMAV — The IMAV bit indicates that a terminated data message or event such as DCAS is in the Input Queue. A terminated message ends in EOI or EOS or with a DCAS or DTAS event.

IDAV — The IDAV bit indicates that data in the Input Queue is not necessarily terminated by EOI or EOS.

IMBF — The IMBF bit indicates that the Input Queue has reached its maximum message buffer size. A finite number of terminated messages, configurable by `MESSAGE_BUFFER_SIZE` in the `io_param.h` file, are allowed in the Input Queue. Data must be read from the Input Queue before any new terminated messages are allowed into the queue.

Note: ***Empty space may exist in the message buffer (!IMBF) while the Input Queue is full (IQF) of data.***

ISAV — The ISAV bit is the analogous status bit of IQF. ISAV is set when there is space for data in the Input Queue.

Note: ***Empty space may exist in the Input Queue (!IQF) while the message buffer is full (IMBF), so no more data is allowed into the Input Queue.***

IQAC — The IQAC bit indicates that the Input Queue has set up a new I/O process; no other ASYNC process is allowed while the Input Queue is active.

IQEN — The IQEN bit indicates that the Input Queue is turned on. You can turn the Input Queue on and off by using `Input_Queue_Enable()` and `Input_Queue_Disable()`.

OQF — The OQF bit indicates that all data space in the Output Queue is used. No more data is received by the Output Queue while OQF is set. Subsequent writes will return an EOQ because no data space is available. You can avoid an EOQ error by using the function `Output_Space_Available()` to indicate how much space is in the Output Queue.

OQE — The OQE bit indicates that no data is left in the Output Queue.

OMWA — The OMWA bit indicates that a terminated data message is in the Output Queue. A terminated message ends in EOI or EOS.

ODWA — The ODWA bit indicates that data in the Output Queue is not necessarily terminated by EOI or EOS.

OMBF — The OMBF bit indicates that the Output Queue has reached its maximum message buffer size. A finite number of terminated messages, configurable by `MESSAGE_BUFFER_SIZE` in the `io_param.h` file, are allowed in the Output Queue. Data must be read from the Output Queue before any new terminated messages are allowed.

Note: ***Empty space may exist in the message buffer (!OMBF) while the Output Queue is full (OQF).***

OSAV — The OSAV bit is the analogous status bit of OQF. OSAV is set when there is space for data in the Output Queue.

Note: ***Empty space may exist in the Output Queue (!OQF) while the message buffer is full (OMBF).***

OQAC — The OQAC bit indicates that the Output Queue has set up a new I/O process; no other ASYNC process is allowed while the Output Queue is active.

OQEN — The OQEN bit indicates that the Output Queue is turned on. You can turn the Output Queue off and on by using `Output_Queue_Enable()` and `Output_Queue_Disable()`.

Count Variable: QUEUE_COUNT

`QUEUE_COUNT` returns the number of bytes read from or written to the queue by the queue I/O calls `Read_Input_Queue()` and `Write_Output_Queue()`.

Address Variable: QUEUE_ADDRESS

If the device program uses multiple addresses while reading from the Input Queue (`Read_Input_Queue()`), `QUEUE_ADDRESS` returns the address for which the data is meant. `QUEUE_ADDRESS` cannot be used like `MULTI_ADDRESS` for outputting data. If you want to write from a specific address, you must disable the Output Queue and use `MULTI_ADDRESS` with the `SEND/SEND_ASYNC()` functions.

Configuring the Queues

The `io_param.h` file contains four important constants to configure the queues with. See Table 4-2.

Table 4-2. io_param.h Queue Configuration Constants

Constant	Valid Values	Description
`USE_QUEUES`	YES/NO	Load the Interrupt Handler for the Queues
`INPUT_QUEUE_SIZE`	0 to 0xffffffff	Length of Input Queue
`OUTPUT_QUEUE_SIZE`	0 to 0xffffffff	Length of Output Queue
`MAXIMUM_BUFFER_SIZE`	0 to 0xffffffff	Buffer Size Should be Smaller than Queue Size
`USE_EOS_WITH_QUEUES`	YES/NO	Detect EOS Bytes Set by `WRITE_EOS_BYTE/ READ_EOS_BYTE`

Queue sizes should hold a reasonably large message in order to make I/O and data handling more efficient. A smaller queue is not a problem, though; data queues can be filled with an unterminated message, then filled again repeatedly. A queue never accepts or transmits data that is larger than the queue size. It is inefficient, however, to have a queue the size of 100 bytes when a typical message is 10,000 bytes.

You can use the Enable or Disable Queue functions to turn individual queues on and off. Making the size of the queue zero does not disable it.

Queue Operation

When the device program starts up, the function `Initialize_Interface()` sets various TNT4882 registers to their default state and loads the main `Interrupt_Handler()`. If `USE_QUEUES` is set to `YES`, the `Initialize_Interface()` function calls

```
Set_Interrupt_Function(R_imr2, B_adsc,
Queue_Interrupt_Handler);
```

which enables the calling of `Queue_Interrupt_Handler` on the address state change interrupt. Whenever the address state of the TNT4882 changes, the `Interrupt_Handler()` function calls `Queue_Interrupt_Handler()`.

Input Queue Handling

When an ADSC interrupt occurs, Queue_Interrupt_Handler() checks its address state for TACS|LACS. If the TNT4882 is addressed to listen, it calculates the empty space in the Input Queue and begins a read of that amount. The USE_DMA configuration option in io_param.h determines the method of transfer. If the Input Queue is full or no space is available for another terminated message, the Input Queue does not perform the read. If data is available in the Input Queue, the IDAV bit is set in QUEUE_STATUS. If a terminated message is in the Input Queue, the IMAV bit is set.

Data is read from the Input Queue by using use Read_Input_Queue(). QUEUE_COUNT indicates the number of bytes read. QUEUE_ADDRESS indicates the address that the data was written to (if using multiple addresses). If the Input Queue is inactive because it is full, Read_Input_Queue() forces another read to occur if the interface is listen addressed. At the end of the read, Queue_Interrupt_Handler() is called again by the I/O clean-up routine, DONE_Interrupt_Handler().

Input Queue and Events

The Input Queue can have events such as device trigger (DTAS) and device clear (DCAS) registered in it. When read from the Input Queue, the event message is contained in the terminator. This terminator indicates whether EOI, EOS, DCAS, or DTAS has occurred. Typically, these events have zero length and are indicated by the IMAV status bit or the MESSAGE_AVAILABLE() function. The io.param.h constant INCLUDE_EVENT_HANDLERS can enable these events. You can use DET_Interrupt_Handler() and DEC_Interrupt_Handler() as models for including more events in the Input Queue. For more information, see the *Event Handling* section in Chapter 6, *Advanced Topics*.

Output Queue Handling

When an ADSC interrupt occurs, Queue_Interrupt_Handler() checks its address state for TACS|LACS. If the TNT4882 is addressed to talk, it checks the Output Queue to see if data will be sent from it. If there is data or a completed message, Queue_Interrupt_Handler() begins a write of the length of the data message. The USE_DMA configuration option in io_param.h determines the method of transfer. If the Output Queue is empty, it does not perform the write. If data is waiting in the Output Queue, the ODWA bit is set in QUEUE_STATUS. If a terminated message is in the Output Queue, the OMWA bit is set.

Use Write_Output_Queue() to write data to the Output Queue. QUEUE_COUNT contains the number of bytes written to the Output Queue. If a queue is inactive because it is empty, Write_Output_Queue() forces another write to occur if the interface is talk addressed. At the end of the write, Queue_Interrupt_Handler is called again by the I/O clean-up function, DONE_Interrupt_Handler().

Queue Function Descriptions

The remainder of this chapter is a quick reference for the TNT4882 ESP-488TL queue functions.

Initialization Functions

Use these functions to initialize the queues.

`Initialize_Input_Queue();`	Sets pointers to the Input Queue to the pointers' default state.
`Initialize_Output_Queue();`	Sets pointers to the Output Queue to the pointers' default state.

Status Functions

These functions return important status information about the queues.

`Output_Space_Available()`	Returns the amount of space available in the Output Queue.
`Message_Available()`	Indicates the IMAV `QUEUE_STATUS` bit.
`Data_Available()`	Indicates the IDAV and IMAV `QUEUE_STATUS` bits.
`Message_Length()`	Indicates how long the next data message in the Input Queue is.
`Wait_For_Queue(int mask)`	Waits for the mask field to be set in `QUEUE_STATUS`.

I/O Functions

These functions control I/O to and from the queues.

```
Write_Output_Queue(char *buf, unsigned long int cnt,
                   int term)
```

Inserts data into the Output Queue.

`Read_Input_Queue(char *buf, unsigned long int cnt, int *term)`
Retrieves data from the Input Queue.

`Disable_Input_Queue()` Turns the Input Queue off.

`Enable_Input_Queue()` Turns the Input Queue on.

`Disable_Output_Queue()` Turns the Output Queue off.

`Enable_Output_Queue()` Turns the Output Queue on.

Chapter 5
TNT4882 Talker/Listener ESP Queue Functions and Utilities Reference

This chapter contains information that explains how to use the queue functions and utilities in the TNT4882 ESP-488TL interrupt package. This chapter lists the functions in alphabetical order.

The function descriptions in this chapter discuss the following information:

- Function Prototype — Lists the prototype of how the function is declared.
- Source File — States where the function is declared.
- Header File — Gives the header file for the function.
- Related Functions — List other associated functions.
- Purpose — Gives a short explanation of the function.
- Description — Gives a detailed description of the function.
- Parameters — List the parameters in the argument list description.
- Equivalents — List the equivalent functions in the source code.
- Examples — Give examples of usage.

int Data_Available()

Source File: `queue_io.h` (macro)
Header File: `queue_io.h`
Related Functions: `Message_Available()`, `Message_Length()`, `Message_Available()`

Purpose

Indicate that data is still in the Input Queue.

Description

You can use `Data_Available()` to determine when a terminated message or an unterminated message is in the Input Queue. An unterminated message shows that data was placed in the Input Queue but that this data was not terminated by EOI, EOS, or an event. An unterminated message usually occurs when the device's buffer is too small for the data that is sent to it: the data buffer fills up before the Talker sends the terminator. The length of the data message is indicated by `MESSAGE_LENGTH()`.

Parameters

None.

Equivalents

```
Data_Available() = QUEUE_STATUS&(IMAV | IDAV);
```

Examples

1. Read data from the Input Queue.

```
if(Data_Available())
   Read_Input_Queue(buf, Message_Length(), &term);
```

If there is data in the Input Queue, read the data.

2. Read at most five bytes from the Input Queue.

```
if(Data_Available())
   Read_Input_Queue(buf, 5, &term);
```

If there is data in the Input Queue, read 5 bytes of data.

void Disable_Input_Queue()

Source File: `queue_io.h` (macro)
Header File: `queue_io.h`
Related Functions: `Initialize_Input_Queue()`, `Initialize_Output_Queue()`, `Enable_Input_Queue()`, `Enable_Output_Queue()`, `Disable_Output_Queue()`

Purpose

Turn the Input Queue off.

Description

`Disable_Input_Queue()` can turn the Input Queue off by disabling its Input Queue Enabled (IQEN) bit and calling `DONE_Interrupt_Handler()` if the Input Queue is active (IQAC). Once the queue is disabled, the `Receive/ Receive_ASYNC()` function can read data from the GPIB.

Parameters

None.

Equivalents

```
Disable_Input_Queue() = Queue_Control(INPUT, DISABLE);
```

Examples

1. Turn the Input Queue off.

```
Disable_Input_Queue();              Disable queue.
```

2. Reinitialize the Input Queue (`io_param.h USE_QUEUES = YES`).

```
void main(void)
{
   Initialize_Interface();          Initialize the TNT4882 and turn
                                    queues on.
   ..........
   ..........
   Disable_Input_Queue();           Turn the Input Queue off.
   Initialize_Input_Queue();        Reinitialize the Input Queue.
   Enable_Input_Queue();            Turn the Input Queue on.
  ..........
}
```

void Disable_Output_Queue()

Source File: `queue_io.h` (macro)
Header File: `queue_io.h`
Related Functions: `Initialize_Input_Queue()`, `Initialize_Output_Queue()`, `Enable_Input_Queue()`, `Disable_Input_Queue()`, `Enable_Output_Queue()`

Purpose

Disable the Output Queue.

Description

`Disable_Output_Queue()` can turn the Output Queue off by disabling its Output Queue Enabled (OQEN) bit and calling `DONE_Interrupt_Handler()` if the Output Queue is active (OQAC). Once the queue is disabled, the `Send/ Send_ASYNC()` function can write data to the GPIB.

Parameters

None.

Equivalents

`Disable_Output_Queue() = Queue_Control(OUTPUT, DISABLE)`

Examples

1 Turn the Output Queue off.

```
Disable_Output_Queue();            Disable queue.
```

2. Reinitialize the Output Queue (`io_param.h USE_QUEUES = YES`).

```
void main(void)
{
   Initialize_Interface();         Initialize the TNT4882 and turn
                                   queues on.
   ..........
   ..........
   Disable_Output_Queue();         Turn the Output Queue off.
   Initialize_Output_Queue();      Reinitialize the Output Queue.
   Enable_Output_Queue();          Turn the Output Queue on.
   ..........
}
```

void Enable_Input_Queue()

Source File: `queue_io.h` (macro)
Header File: `queue_io.h`
Related Functions: `Initialize_Input_Queue()`, `Initialize_Output_Queue()`, `Disable_Input_Queue()`, `Enable_Output_Queue()`, `Disable_Output_Queue()`

Purpose

Enable the Input Queue.

Description

`Enable_Input_Queue()` turns the Input Queue on by enabling IQEN and calling `Queue_Interrupt_Handler()` if the TNT4882 is listen addressed.

Parameters

None.

Equivalents

`Enable_Input_Queue() = Queue_Control(INPUT, ENABLE)`

Examples

1. Turn the Input Queue on.
```
Enable_Input_Queue();
```

2. Reinitialize the Input Queue (`io_param.h USE_QUEUES = YES`).
```
void main(void)
{
   Initialize_Interface();       Initialize the TNT4882 and turn
                                 queues on.
   ..........
   ..........
   Disable_Input_Queue();        Turn the Input Queue off.
   Initialize_Input_Queue();     Reinitialize the Input Queue.
   Enable_Input_Queue();         Turn the Input Queue on.
   ..........
}
```

void Enable_Output_Queue()

Source File: `queue_io.h` (macro)
Header File: `queue_io.h`
Related Functions: `Initialize_Input_Queue()`, `Initialize_Output_Queue()`, `Enable_Input_Queue()`, `Disable_Input_Queue()`, `Disable_Output_Queue()`

Purpose

Enable the Output Queue.

Description

`Enable_Output_Queue()` turns the Output Queue on by enabling OQEN and calling `Queue_Interrupt_Handler()` if the TNT4882 is talk addressed.

Parameters

None.

Equivalents

`Enable_Output_Queue() = Queue_Control(OUTPUT, ENABLE)`

Examples

1. Turn the Output Queue on.
   ```
   Enable_Output_Queue();
   ```

2. Reinitialize the Output Queue (`io_param.h USE_QUEUES = YES`).
   ```
   void main(void)
   {
      Initialize_Interface();       Initialize the TNT4882 and turn
                                    queues on.
      ..........
      ..........
      Disable_Output_Queue();       Turn the Output Queue off.
      Initialize_Output_Queue();    Reinitialize the Output Queue.
      Enable_Output_Queue();        Turn the Output Queue on.
      ..........
   }
   ```

void Initialize_Input_Queue()

Source File: `queue_io.h` (macro)
Header File: `queue_io.h`
Related Functions: `Initialize_Output_Queue()`, `Enable_Input_Queue()`, `Enable_Output_Queue()`, `Disable_Input_Queue()`, `Disable_Output_Queue()`

Purpose

Set queue pointers to the Input Queue to their default state.

Description

`Initialize_Input_Queue()` sets the in, out, and top queue buffer pointers to point to the top of the Input Queue buffer space and sets the bottom pointer to point to the bottom of the Input Queue space. When the pointers are initialized, the Input Queue is reinitialized. The Input Queue is disabled until `Enable_Input_Queue()` is executed. `Initialize_Interface()` originally calls `Initialize_Input_Queue()`.

Parameters

None.

Equivalents

`Initialize_Input_Queue() = Initialize_Queue(INPUT)`

Example

1. Reinitialize the Input Queue (`io_param.h USE_QUEUES = YES`).

```
void main(void)
{
   Initialize_Interface();       Initialize the TNT4882 and turn
                                 queues on.
   ..........
   ..........
   Disable_Input_Queue();        Turn the Input Queue off.
   Initialize_Input_Queue();     Reinitialize the Input Queue.
   Enable_Input_Queue();         Turn the Input Queue on.
   ..........
}
```

void Initialize_Output_Queue()

Source File: `queue_io.h` (macro)
Header File: `queue_io.h`
Related Functions: `Initialize_Input_Queue()`, `Enable_Input_Queue()`, `Enable_Output_Queue()`, `Disable_Input_Queue()`, `Disable_Output_Queue()`

Purpose

Set pointers to the Output Queue to their default state.

Description

`Initialize_Output_Queue()` sets the in, out, and top queue buffer pointers to point to the top of the Output Queue buffer space and sets the bottom pointer to point to the bottom of the Output Queue space. When the pointers are initialized, the Output Queue is reinitialized, but the Output Queue is disabled until `Enable_Output_Queue()` is executed. `Initialize_Interface()` originally calls `Initialize_Output_Queue()`.

Parameters

None.

Equivalents

`Initialize_Output_Queue() = Initialize_Queue(OUTPUT)`

Example

1. Reinitialize the Output Queue (`io_param.h USE_QUEUES = YES`).

```
void main(void)
{
   Initialize_Interface();        Initialize the TNT4882 and turn
                                  queues on.
   ..........
   ..........
   Disable_Output_Queue();        Turn the Output Queue off.
   Initialize_Output_Queue();     Reinitialize the Output Queue.
   Enable_Output_Queue();         Turn the Output Queue off.
   ..........
}
```

int Message_Available()

Source File: `queue_io.h` (macro)
Header File: `queue_io.h`
Related Functions: `Message_Length()`, `Data_Available()`

Purpose

Indicate that at least one terminated message or event is in the Input Queue.

Description

You can use `Message_Available()` to determine when a terminated message is in the Input Queue. A terminated message can be either a terminated data message or an event. A terminated data message is terminated with EOI or EOS and has a length; an event is a multiline interface message, such as Device Trigger (DTAS) or Device Clear (DCAS), and normally has a length of zero.

Parameters

None.

Equivalents

```
Message_Available() = QUEUE_STATUS&IMAV;
```

Examples

1. Read data from the Input Queue.

```
if(Message_Available()){                    Check for a message.
   Read_Input_Queue(buf, Message_Length(), &term);
                                            Read message.
   My_Parser(buf, QUEUE_COUNT, term);
                                            Send data to the parser; the size
                                            of the data in the buffer is
                                            QUEUE_COUNT.
}
```

2. Check for an event.

```
if(Message_Available()){                    Check for a message.
   Read_Input_Queue(buf, INPUT_QUEUE_SIZE, &term);
                                            Return terminated message.
   if(term&(DCAS | DTAS)){                  If it has an event after it.
      Event_Handler(term);                  Handle the event.
   }
}
```

unsigned long int Message_Length()

Source File: `queue_io.h` (macro)
Header File: `queue_io.h`
Related Functions: `Message_Length(), Message_Available(), Data_Available();`

Purpose

Indicate how long the next message in the queue is.

Description

This function always returns the length value of the queue message structure for the current available terminated/unterminated message.

Parameters

None.

Equivalents

```
Message_Length() =
Input_Queue_Info.message[Input_Out_Msg_Index].length;
```

Examples

1. Get the current message length.

```
curr_length = Message_Length();
```

Return the length of the current message.

2. Read data from the Input Queue.

```
if(Message_Available()){                Check for a message.
   Read_Input_Queue(buf, Message_Length(), &term);
                                        Read message.
   Parser(buf,QUEUE_COUNT,term);
                                        Send data to the parser; the size
                                        of the data in the buffer is
                                        QUEUE_COUNT.
}
```

unsigned long int Output_Space_Available()

Source File: `queue_io.h` (macro)
Header File: `queue_io.h`
Related Functions: `Write_Output_Queue()`

Purpose

Return the amount of empty data space that is available in the Output Queue for data input.

Description

This function determines the amount of empty space left in the Output Queue. If the Output Message Buffer Full (OMBF) or Output Queue Full (OQF) bit is set in `QUEUE_STATUS`, this function returns a zero.

Parameters

None.

Equivalents

None.

Examples

1. Write a continuous stream of data to the Output Queue.

```
while(1){                                   Loop forever.
   cnt = Output_Space_Available()
                                            Get space available.
   Write_Output_Queue(buf, cnt, NONE);
                                            Write data to the Output Queue.
}
```

2. Write a data block that is larger than the Output Queue to the Output Queue.

```
while(original_cnt!=0){                     Loop until the write is
                                            complete.
   temp_cnt = Output_Space_Available();
                                            Get space available.
   if (temp_cnt<original_cnt){              If space is smaller than the data
                                            left to be output.
      term = NONE;                          Do not terminate.
      cnt = temp_cnt;                       cnt = space available.
   }
   else{                                    Else.
      term = EOI;                           Terminate with EOI.
      cnt = original_cnt;                   cnt = data count left.
   }
```

```
    if (cnt){                               If there is data to be written,
    write data.
        Write_Output_Queue(buf, cnt, term);

        original_cnt- = QUEUE_COUNT;
                                            Update original_cnt.
        buf + = QUEUE_COUNT;                Update the buffer.
    }
}
```

void Read_Input_Queue(char_buf *buf, unsigned long int cnt, int *term)

Source File: `queue_io.c`
Header File: `queue_io.h`
Related Functions: `Message_Length()`, `Message_Available()`, `Write_Output_Queue()`, `Data_Available()`

Purpose

Read data from the Input Queue.

Description

`Read_Input_Queue()` reads the amount of bytes indicated by `cnt` from the data buffer of the Input Queue and copies them to the location pointed to by `buf`. If `cnt` exceeds the number of bytes available in the terminated message, `Read_Input_Queue()` reads only the number of bytes available. The number of bytes read is contained in `QUEUE_COUNT`. The message read from the Input Queue is terminated by `term`. The termination can be EOI, EOS, NONE, or an event like DCAS or DTAS—if these event interrupt handlers are enabled (see the *Input Queue and Events* section in Chapter 4, *Queues*). If multiple addressing is used, the global value `QUEUE_ADDRESS` is updated every time the `Read_Input_Queue()` function is implemented. `QUEUE_ADDRESS` represents the address the TNT4882 was recognizing when it read the message.

Parameters

`buf` = character pointer to destination of data to be read from the Input Queue data buffer.

`cnt` = 0 to 0xffffffff (4G)	Number of bytes that must be read from the queue.
`term` = EOI (END, 0x2000)	EOI with the last byte received.
= EOS (0x1000)	EOS byte received.
= NONE (0)	No termination.
= DTAS (0x10)	Device Trigger received.
= DCAS (0x08)	Device Clear received.

Equivalents

None.

Examples

1. Read up to 100 bytes into the "buf" from the Input Queue.
```
Read_Input_Queue(buf, 100, &term);
```

2. Read data from the Input Queue.

```
if(Message_Available()){          Check for a message.
   Read_Input_Queue(buf, Message_Length(), & term);
                                  Read message.
                                  Send data to the parser; the size
                                  of the data in the buffer is
                                  QUEUE_COUNT.
   My_Parser(buf, QUEUE_COUNT, term);

   }
```

3. Check for an event.

```
if(Message_Available()){          Check for a message.
                                  Read terminated message; the
                                  size of the message is in
                                  QUEUE_COUNT.
   Read_Input_Queue(buf, INPUT_QUEUE_SIZE, & term);

   if(term&(DCAS | DTAS)){
      Event_Handler(term);        If terminator contains an event,
      handle the event.
   }
}
```

4. Multiple Addresses.

```
io_param.h file settings
USE_SECONDARY_ADDRESSES = YES
SECONDARY_ADDRESSES{4, 17, 20}
PRIMARY_ADDRESS 5
If (Message_Available()),
                                  Read a message from the Input
   Queue.
   Read_Input_Queue(buf, 1000, & term)

switch(QUEUE_ADDRESS){            Do something for a specific
                                  address.
   case 0x0504:
      ..........
      break;
   case 0x0511:
      ..........
      break;
   case 0x0514:
      ..........
      break;
}
```

void Wait_For_Queue(int mask)

Source File: `queue_io.c`
Include File: `queue_io.h`
Related Functions: None

Purpose

Wait until the mask event occurs in the `QUEUE_STATUS` status word.

Description

`Wait_For_Queue()` compares `QUEUE_STATUS` to the mask. When any condition in the mask is true, the call exits and the program continues. The `mask` variable relates to the status bits in Table 5-1.

Table 5-1. QUEUE_STATUS Status Bits

Mnemonic	Bit Position	Description
IQF	15	Input Queue Full
IQE	14	Input Queue Empty
IMAV	13	Input Message/Event Available
IDAV	12	Input Data Bytes Available
IMBF	11	Input Message Buffer Full
ISAV	10	Input Space Available
IQAC	9	Input Queue Active
IQEN	8	Input Queue Enabled
OQF	7	Output Queue Full
OQE	6	Output Queue Empty
OMWA	5	Output Message Waiting
ODWA	4	Output Data Bytes Waiting
OMBF	3	Output Message Buffer Full
OSAV	2	Output Space Available
OQAC	1	Output Queue Active
OQEN	0	Output Queue Enabled

Parameters

`mask` = 0x00 to 0xff — Mask related to `QUEUE_STATUS` bits.

Equivalents

None.

Examples

1. Wait for any input data.
   ```
   Wait_For_Queue(IMAV | IDAV);
   ```

2. Wait for available output space.
   ```
   Wait_For_Queue(OSAV);
   ```

void Write_Output_Queue(char_buf *buf, unsigned long int cnt, int term)

Source File: `queue_io.c`
Header File: `queue_io.h`
Related Functions: `Message_Length()`, `Message_Available()`, `Read_Input_Queue()`, `Output_Space_Available()`

Purpose

Insert data into the Output Queue.

Description

`Write_Output_Queue()` places a number of bytes indicated by `cnt` and pointed to by `buf` into the data buffer of the Output Queue. If the value of `cnt` exceeds what the Output Queue can hold, `Write_Output_Queue()` writes as many bytes as it can fit into the Output Queue. The number of bytes written to the Output Queue is the value contained in `QUEUE_COUNT`. The message written to the Output Queue is terminated by the method described by `term`.

Parameters

`buf` = character pointer to data to be written to the Output Queue buffer.

`cnt` = 0 to 0xffffffff (4G)	Number of bytes that must be written to the Output Queue.
`term` = EOI (END, 0x2000)	Set EOI with the last byte.
= EOS (0x1000)	Set EOI with the EOS byte.
= NONE (0)	No termination.

Equivalents

None.

Examples

1. Write a continuous stream of data to the Output Queue.

```
while(1) {                                  Loop forever.
    cnt = Output_Space_Available();
                                            Get space available.
    Write_Output_Queue(buf,cnt,NONE);
                                            Write data to the Output Queue.
}
```

2. Use the EOS byte on the write.

```
io_param.h WRITE_EOS_BYTE = 0x0a
Write_Output_Queue("Data abcdefg \n", 14, EOI | EOS);
```

Write data to the queue and set for EOS termination.

3. Write a data block that is larger than the Output Queue to the Output Queue

```
while(original_cnt! = 0) {          Loop until the write is
                                    complete.
   temp_cnt = Output_Space_Available();
                                    Get space available.
   if (temp_cnt<original_cnt) {
                                    If space is smaller than data left
                                    to be output.
      term = NONE;                  Do not terminate.
      cnt = temp_cnt;               cnt = space available.
   }
   else {                           Else.
      term = EOI;                   Terminate with EOI.
      cnt = original_cnt;           cnt = data count left.
   }
   if (cnt) {
      Write_Output_Queue(buf, cnt, term);
                                    If there is data to be written,
         write data.
      original_cnt -  = QUEUE_COUNT;
                                    Update the original_cnt.
      buf + = QUEUE_COUNT;          Update buf.
   }
}
```

Chapter 6
Advanced Topics

This chapter explains TNT4882 programming fundamentals and issues that you need to be aware of when you rewrite or change the ESP code.

TNT4882 Initialization

Complete the steps below to initialize the TNT4882 to a default power-on state. You can complete these steps by looking at the `Initialize_Interface()` source code.

Initialization Steps

See `Initialize_Interface()` in `ngpib_io.c` or `gpib_io.c`.

1. Issue a software reset by writing `0x22`, Soft Reset, to the Command Register (CMDR).

2. Place the TNT4882 in Turbo+7210 mode by

   - Writing `0x80` to SPMR
   - Writing `0x80` to AUXMR
   - Writing `0x99` to AUXMR
   - Writing `0x00` to KEYREG

   This sequence places the TNT4882 into Turbo+7210 mode, no matter how the chip is configured (other configurations are the 9914 and the swapped 9914 register map).

3. Set the handshake mode by writing to the Handshake Select Register (HSSEL). You must set the handshake mode and the Data Valid (DAV) deglitching value (in HIER) even if you do not want high-speed transfers. (The ESP code is written to use the TNT4882 in one-chip mode.) See the *Configuring the TNT4882 Handshake Mode and High-Speed I/O* section, which is located later in this chapter.

4. Issue a chip reset by writing `0x02`, Chip Reset, to the Auxiliary Mode Register (AUXMR). This action sets the local pon message. The TNT4882 ignores all GPIB activity until pon is cleared.

5. Write the desired status byte to the Serial Poll Mode Register (SPMR).

6. Write to the hidden Parallel Poll Register (PPR) if locally programming for parallel polls.

7. Select the desired addressing mode by writing to the Address Mode Register (ADMR).

8. Write the desired primary GPIB address by writing to the Address Register (ADR).

9. Write the desired secondary GPIB address by writing to the ADR if the address mode includes a secondary address.

10. Write the desired AUXMR setups if common configuration is noted.

    - AUXRA — Use for I/O (not commonly written to for initialization).

    - AUXRB — Set the Three-State Timing (TRI) bit for high-speed T1 delay. T1 is the delay time that occurs before the first data byte is sourced. If you are using HS488, you must detect the 0x1f High-Speed Configuration command, so set the Command Pass Through Enable (CPT ENAB) bit to enable the TNT4882 to detect unknown commands.

    - AUXRE — Use to hold off the handshake in case of Device Clear or Trigger (not commonly written to).

    - AUXRF — Use to hold off the handshake in case of a Talk Address, Listen Address, Untalk, and Unlisten (not commonly written to).

    - AUXRG — Use for two-chip mode settings (not commonly written to).

    - AUXRI — Set the Ultra Short T1 Delay (USTD) bit if you want a short T1 delay. Set the Static Interrupt Status bits (SISBs), or *static bits,* to make the interrupt status registers (ISR0, ISR1, and ISR2) clear interrupt bits when certain conditions are met instead of clear interrupt bits that are read. Setting the SISBs helps keep a copy of an interrupt condition in the status register until the status condition can be properly serviced or read. You clear interrupt conditions by issuing auxiliary commands, such as `F_clrERR` (`0x57`), to the AUXMR.

    - AUXRJ — Use to set timeout time (not commonly written at initialization time).

11. Write the Holdoff Immediately auxiliary command, `0x51`, to the AUXMR to prevent data reception.

12. Issue power on`, 0x00`, to the AUXMR to clear power-on that was set by the chip reset.

13. Set any desired interrupt masks. Common interrupts for a device are DET, DEC, and ADSC.

Configuring the TNT4882 Handshake Mode and High-Speed I/O

The TNT4882 can perform both the normal, interlocked, three-wire 488 GPIB handshake and HS488 noninterlocked, high-speed handshakes. Hardware detects the HS488 capabilities of communicating devices, so software does not have to switch between the three-wire GPIB handshake and high speed: only an initial configuration is necessary.

The Controller enables the HS488 capabilities of the TNT4882 by sending two GPIB commands. The first byte is 0x1f, high-speed Configure Enable (CFE). The second byte, which corresponds to the physical length of the cable in the GPIB system, is a secondary command byte of the form 0x6*x*, where *x* [1, 2, 3, .. E, F] is the physical length (in meters) of the cable in the GPIB system. The Controller needs to send these commands once. The TNT4882 recognizes the sequence of these command bytes as unknown commands. With CPT ENAB set in Auxiliary Register B (AUXRB), the CPT Interrupt Status bit of Interrupt Status Register 1 (ISR1) sets when an unknown command is received. The unknown command can then be read from the Command Pass Through Register (CPTR). The CPT Handler functions manage this process.

The process outlined in the following *Detecting the CF Command* section demonstrates the steps you can take to detect the HS488 command bytes CFE and CFGn (n = [1, 2, ... 15]). You can follow these steps by looking at the `CPT_Handler()` and `Validate_CF_Command()` ESP codes.

Detecting the CF Command

1. Set the CPT ENAB bit initially in the AUXRB.

2. Perform step 3 when the CPT bit in ISR1 is set.

3. Read the CPTR and compare its value with the HSC value of 0x1f. If the CPTR value is 0x1f, set a software flag that recognizes that the HSC command has been received. Otherwise, clear the flag.

4. Perform step 5 when the CPT bit in ISR1 is set again.

5. If the flag recognizing the previous command as HSC is set, read the CPTR and compare its value to 0x6x. If the CPTR value is 0x6*x*, configure the timing registers for the corresponding *x* length and clear the flag. If the CPTR value is not 0x6*x*, clear the software HSC flag.

6. Use the *x* cable length value to set up the TNT4882 timing values according to Table 6-1, then turn on high-speed capability. See the following *Configuring the TNT4882 Mode and Timing Registers* section.

Table 6-1. Timing Registers Configuration Table

Cable Lengths	**Factor T11**	**Factor T12**	**Deglitch**	**Maximum Transfer Rate in Hz**
1	`NO_TSETUP`	0	0	8 M
2	`NO_TSETUP`	0	0	8 M
3	0	2	0	5 M
4	0	2	DGA	5 M
5	2	4	DGA	3.3 M
6	2	4	DGA	3.3 M
7	2	4	DGA	3.3 M
8	4	6	DGA|DGB	2.5 M
9	4	6	DGA|DGB	2.5 M
10	6	8	DGA|DGB	2 M
11	6	8	DGA|DGB	2 M
12	6	8	DGA|DGB	2 M
13	9	11	DGA|DGB	1.5 M
14	9	11	DGA|DGB	1.5 M
15	9	11	DGA|DGB	1.5 M

Configuring the TNT4882 Mode and Timing Registers

A few configuration registers must be initialized to set up the TNT4882 handshake and timing that are used in high-speed situations. The handshake mode can be set in the HSSEL. Even if you will not use high-speed transfers, you must set the TNT4882 into one-chip mode and set the DAV deglitching circuits.

The TNT4882 used to be essentially two chips: the Turbo488 and the NAT4882. The TNT4882 can be accessed either like the Turbo488 and the NAT4882 or like one chip. The one-chip mode is an optimization of the two-chip combination. The timing registers are hidden registers located in the SH_CNT Register, Source Handshake Count. Even if you do not plan to use the high-speed capabilities of the TNT4882, you must still set the deglitching values in the High-Speed Enable Register (HIER). The timing values can vary, depending on the length of the cable attached to the entire GPIB system (see Table 6-1). Because it is possible to obtain a higher throughput on a shorter cable, these values have been optimized for shorter cable lengths. Although a system may have *x* m of cable, any timing value of *x* or more meters will work in the system (*x* is a value between 0 and 15).

Three timing values—T11, T12, and T17—are defined as follows:

- T11 Delay until a DAV assertion (setup there).
- T12 Duration of DAV assertion.
- T17 NRFD wink duration.

The steps outlined in the following section demonstrate the process of configuring the TNT4882 mode and timing registers. You can follow this process by looking at the `High_Speed_Select()` ESP code.

Setting the TNT4882 to the Normal Three-Wire Handshake Mode

1. Write `0x01` to the HSSEL to select one-chip mode.
2. Set the Deglitch Selector A and Deglitch Selector B (DGA and DGB) bits in the HIER.
3. Clear the High-Speed Enable (HSE) bit in the Miscellaneous Register (MISC).

Setting the TNT4882 to HS488

1. Write `0x01` to the HSSEL to select one-chip mode.
2. Write to the SH_CNT to set T11 (TSETUP). TSETUP is a hidden register with a T11 binary pattern of [110xxxxx], where the *x's* represent the binary value for the factor.
3. Write to the HIER. Set the NO TSETUP bit if T11 in the chart says `NO_TSETUP`. Set the deglitching bits (DGA and DGB) according to Table 6-1.

4. Write to the SH_CNT to set T12. T12 is a hidden register with a binary pattern of [100xxxxx], where the *x's* represent the binary value for the T12 factor.

5. Write to the SH_CNT to set T17. T17 is a hidden register with a binary pattern of [010xxxxx], where the *x's* represent the binary value for the factor. The factor is 0x12, which corresponds to 500 ns, for the T17 register.

6. Write to the MISC to set the HSE bit.

Addressing Modes and Detecting the Address State

The TNT4882 can be programmed to detect six different addressing modes:

- Single primary (normal)
- Single primary–single secondary (extended)
- Single primary–multiple secondary (multiple extended)
- Listen only/talk only
- Multiple primary
- Multiple primary and multiple secondary

These modes are discussed in detail in the following sections.

Single Primary Addressing

Single primary addressing mode is the most common addressing mode among IEEE 488.2 devices. In single primary addressing mode, the TNT4882 responds to only one primary talk or listen address. To configure the TNT4882 to be addressed in this manner, you must complete the steps outlined in the following *Configuring for Single Primary Addressing* section.

Configuring for Single Primary Addressing

See `Set_Address_Mode()` and `Change_Primary_Address()` in `ngpib_io.c` or `gpib_io.c`.

1. Place the TNT4882 into normal primary addressing mode by writing `0x31` to the ADMR.

2. Write the desired primary address to the ADR.

3. Write `0xe0` to the ADR to disable the minor address for single primary addressing mode. This action sets the Address Register Select (ARS) bit for the minor address and sets the Disable Talker and Disable Listener bits to disable the minor address.

4. Detect the address state by checking the Address Status Register (ADSR) to determine whether the Talker Active (TA) or Listener Active (LA) bit is set.

Single Primary–Single Secondary Addressing

Single primary–single secondary addressing mode is probably the second most common addressing mode among IEEE 488.2 devices. In single primary–single secondary addressing mode, the TNT4882 responds to one primary talk or listen address that is followed by a secondary address. To configure the TNT4882 to be addressed in this manner, you must complete the steps outlined in the following *Configuring for Single Primary–Single Secondary Addressing* section.

Configuring for Single Primary–Single Secondary Addressing

See `Set_Address_Mode () Change_Primary_Address() Change_Secondary_Address()` in `ngpib_io.c` or `gpib_io.c`.

1. Place the TNT4882 into normal primary extended addressing mode by writing `0x32` to the ADMR.

2. Write the desired primary address to the ADR.

3. Write the desired secondary address to the ADR. To set this secondary address, the ARS bit in the ADR must also be set.

4. Detect the address state by checking the ADSR to determine whether the TA or LA bit is set.

Single Primary–Multiple Secondary Addressing

Single primary–multiple secondary addressing mode is not common among IEEE 488.2 devices. In single primary–multiple secondary addressing mode, the TNT4882 responds to one primary talk or listen address that is followed by a secondary address. An auxiliary command must verify that this primary talk or listen address is a legitimate address. To configure the TNT4882 to be addressed in this manner, you must complete the steps outlined in the following *Configuring for Single Primary–Multiple Secondary Addressing* section.

Configuring for Single Primary–Multiple Secondary Addressing

See `Set_Address_Mode() Change Primary_Address() Change_Multiple_Address() Validate_Secondary_Address()` in `ngpib_io.c` or `gpib_io.c`.

1. Place the TNT4882 into multiple extended addressing mode by writing `0x33` to the ADMR.

2. Write the desired primary address to the ADR.

3. Write `0xe0` to the ADR to disable the minor address for single primary–multiple secondary addressing mode. This action sets the ARS bit for the minor address and sets the Disable Talker and Disable Listener bits for this minor address.

4. Have a program read the CPTR when the APT bit is set. In single primary–multiple secondary addressing mode, the ISR1 APT bit is set when a secondary address is received.

   - If the address is correct, issue the Valid auxiliary command (`0x0f`) to the AUXMR.

   - If the address is incorrect, issue the Nonvalid auxiliary command (`0x07`) to the AUXMR.

5. Detect the address state by checking the ADSR to determine whether the TA or LA bit is set.

Listen-Only/Talk-Only Addressing

In the listen-only and talk-only modes, the device program can address the TNT4882. The TNT4882 does not require an External Controller to address it to talk or listen. The address commands of the Controller are simply ignored in listen-only and talk-only mode. Normally, the listen-only and talk-only modes are used for systems without a Controller. To configure the TNT4882 to be addressed to listen only or talk only, you must complete the steps outlined in the following *Configuring for Talk-Only/Listen-Only Addressing* section.

Configuring for Talk-Only/Listen-Only Addressing

See `Set_Address_Mode` in `ngpib_io.c` or `gpib_io.c`.

Talk Only

1. To place the TNT4882 into talk-only mode, set the Talk-Only (TON) bit by writing `0xb0` to the ADMR.

2. To disable the talk-only state, write `0x00` (`0x30` for the AT board) to the ADMR to clear the TON bit, then write the Local Untalk Command (`0x0b`) to the AUXMR.

3. To detect the address state, check the ADSR to determine whether the TA or LA bit is set.

Listen Only

1. To place the TNT4882 into listen-only mode, set the Listen-Only (LON) bit by writing `0x40` to the ADMR.

2. To disable the listen-only state, write `0x00` (`0x30` for the AT board) to the ADMR to clear the LON bit, then write the Local Unlisten command (`0x0c`) to the AUXMR.

3. To detect the address state, check the ADSR to determine whether the TA or LA bit is set.

Multiple Primary Addressing

In multiple primary addressing mode, the TNT4882 can recognize many different primary talk and listen addresses. You can use multiple primary addressing to let the TNT4882 interface act like a gateway to other devices in a chassis-style system. To configure the TNT4882 to be addressed in this manner, you must complete the steps outlined in the following *Configuring for Multiple Primary Addressing* section.

Configuring for Multiple Primary Addressing

See `SET_Address_Mode Change_Multiple_Address () Validate_Primary_Address ()` in `ngpib_io.c` or `gpib_io.c`.

1. Place the TNT4882 into no-address mode by writing `0x30` to the ADMR.

2. Write to Auxiliary Register F (AUXRF) and set the DHATA and DHALA bits. This action sets the Command Pass Through (CPT) bit in ISR1 when a talk or listen address is received.

3. Read the CPTR when the CPT bit of ISR1 is set. Compare the talk or listen address read to the valid addresses.

4. Issue a Valid auxiliary command to the AUXMR, even if the address is incorrect. This action lets the handshake continue.

5. If the talk address is correct, toggle the TON bit in the ADMR by writing `0x70` to the ADMR, then write `0x30` to the ADMR. If the listen address is correct, toggle the LON bit in the ADMR by writing `0x70` to the ADMR, then write `0x30` to the ADMR. Toggling these bits places the TNT4882 into the Talk or Listen Address state.

6. To detect the address state, check the ADSR to determine whether the TA or LA bit is set.

Multiple Primary and Multiple Secondary Addressing

In multiple primary and multiple secondary addressing mode, the TNT4882 can recognize many different primary talk and listen addresses and the addresses' associated secondary addresses. You can use multiple primary and multiple secondary addressing mode to let the TNT4882 interface act like a gateway device to other devices in a chassis-style system. This behavior is similar to multiple primary addressing mode, except that in multiple primary and multiple secondary addressing mode, the next byte must be checked for the proper secondary address after the primary address has been received. To configure the TNT4882 to be addressed in this manner, you must complete the steps outlined in the following *Configuring for Multiple Primary and Multiple Secondary Addressing* section.

Configuring for Multiple Primary and Multiple Secondary Addressing

1. Place the TNT4882 into no-address mode by writing `0x30` to the ADMR.

2. Write to the AUXRF and set the DHATA and DHALA bits. This action sets the CPT bit of ISR1 when a primary talk or listen address is received.

3. Read the CPTR when the CPT bit of ISR1 is set. Compare the talk or listen address read to the valid addresses.

4. If the talk or listen address is correct, set the DHALL, DHATA, and DHALA bits in the AUXRF. This action lets one byte be read at a time, so it is possible to verify the following secondary address. If the address is not correct, do not set DHALL.

5. Issue the Valid auxiliary command (`0x0f`) to the AUXMR, even if the address is incorrect. Issuing the Valid auxiliary command lets the TNT4882 begin handshaking again. If the address is incorrect, do not continue with steps 6, 7, 8, and 9, but return to step 2.

6. Have the next command message set the CPT bit because the DHALL bit is set. When the CPT bit is set, check the CPTR for a valid secondary address for the previous primary address. The secondary address should be the byte following the primary address.

7. Change the AUXRF back to the DHALA and DHATA bits, then issue the Valid auxiliary command (`0x0f`) even if the address is incorrect. This action lets the TNT4882 begin handshaking again.

8. If the primary and secondary addresses received are valid, toggle either the TON or LON bit in the ADMR by writing `0x60` or `0x70`, respectively, to the ADMR, then write `0x30` to the ADMR.

9. Detect the address state by checking the ADSR to determine whether the TA or LA bit is set.

GPIB I/O with the TNT4882

The TNT4882 ESP-488TL package contains a general I/O function, `User_GPIB_IO()`, that can set up reads or writes by using either DMA or programmed I/O. `User_GPIB _IO()` is interrupt driven in the `gpib_io.c` source file and noninterrupt driven in the `ngpib_io.c` file. Follow the source code for this function when you read the following sections on IO configurations.

Setting Up the TNT4882 for I/O: Setup_TNT_IO()

With one exception—which occurs in the DMA case—you use identical steps to set up the TNT4882 for I/O for both DMA and programmed I/O. You follow these steps even if you do not use interrupts. For DMA, however, the Timer Register (TIMER) should be set with a twos complement timeout value in order to force the DRQ signal to unassert after the time value elapses. This feature releases the bus, so the DMA Controller can perform a memory refresh. The timeout value is in 100-ns clock periods. For PCs, the limit for holding the bus is 15 μs, or 0x95 (0x6a twos complement) 100-ns clock periods.

To set up I/O, complete the steps outlined in the following *Set Up the TNT4882 to Input Data* and *Set Up the TNT4882 to Output Data* sections. You can follow these steps by looking at the `Setup_TNT_IO()` ESP source code.

Set Up the TNT4882 to Input Data

See the `Setup_TNT_IO()` source code listing.

1. (If using DMA) Set the TIMER to `0x6a`, so the TNT4882 will release the bus for memory refreshes. This is done in `Initialize_Interface()`.

2. Reset the FIFOs by setting the Reset FIFO (RESET FIFO) bit in the CMDR.

3. Write the twos complement count to the Count Registers (CNT0, CNT1, CNT2, and CNT3) of the number of bytes to be read.

4. Set up the timeout timer register (AUXRJ) by writing a timeout factor value to it, if you are using timeouts. Set the Timeout (TO) bit and the Byte Timeout (BTO) bit in Interrupt Mask Register 0 (IMR0). (See the `Set_Timeout()` source code.)

5. Set the End bit in Interrupt Mask Register 1 (IMR1) to stop transfer on the reception of EOI or EOS.

6. Write the EOS byte in the End-of-String Register (EOSR) if you are using the EOS byte.

7. Set the Holdoff On End (HLDE) bit in Auxiliary Register A (AUXRA). In addition, set the HLDE bit in the AUXRA if you are using an EOS byte. Set the End On EOS (ENDONEOS) bit in the AUXRA. At the end of every input, the TNT4882 must be heldoff to prevent more data reception.

8. Set the following bits in the Configuration Register (CFG):

   - IN Input GPIB Data.
   - TLCHTE Halt on a Talker/Listener Interrupt (ERR, END).
   - TMOE (If using DMA) Limit the Duration of a DMA Burst.
   - TIMBYTN (If using DMA) Assert the DMA Signal for the Length of Time Indicated by the Timer Register.
   - 16BIT (If using word transfers) The 16-bit bit sets up Word Transfers from the

9. Set the GO BIT in the CMDR.

10. Write the Release Holdoff command (`0x03`, rhdf) to the AUXMR.

Set Up the TNT4882 to Output Data

See the `Setup_TNT_IO()` source code listing.

1. (If using DMA) Set the TIMER to `0x6a`, so the TNT4882 will release the bus for memory refreshes. This is done in `Initialize_Interface()`.

2. Reset the FIFOs by setting the RESET FIFO bit in the CMDR.

3. Write the twos complement count of the number of bytes to be written to CNT0, CNT1, CNT2, and CNT3.

4. Set up the AUXRJ by writing a timeout factor value to it, if you are using timeouts. Set the TO bit and the BTO bit in IMR0. (See the `Set_Timeout()` source code.)

5. Set the Error (ERR) bit in IMR1 to stop the transfer on a handshake error.

6. If you are using an EOS byte, write the EOS byte to the EOSR.

7. Set the Holdoff On All Data (HLDA) bit in AUXRA, and if you are using an EOS byte, set the Transmit EOI With EOS (XEOIWEOS) bit.

8. Set the following bits in the CFG. Not setting IN means this is an output setup.

   - TLCHTE Halt on a Talker/Listener Interrupt (ERR, END).
   - TMOE (If using DMA) Limit the Duration of a DMA Burst.
   - TIMBYTN (If using DMA) Assert the DMA Signal for the Length of Time Indicated by Timer Register.
   - 16BIT (If using word transfers to the FIFOs) The 16-bit bit sets up Word Transfers from the FIFOs.
   - CCEN Turn on Carry Cycle to Assert EOI at End of Transfer. The TNT4882 Automatically sends EOI with the last byte of the transfer.

8. Write the Holdoff Immediate command (`0x51`) to the AUXMR.

9. Set the GO bit in the CMDR.

Interrupt and Noninterrupt Programmed I/O: GPIB_PROG_IO()

Programmed I/O is a method of reading from or writing to the FIFOs of the TNT4882 without using the DMA Controller. With this method, I/O must be handled programmatically by periodically checking the Interrupt Status Register to determine what to do.

Important I/O Status Bits

Interrupt Status Register 3 (ISR3) contains three bits that indicate the status of the FIFOs and two bits that indicate the status of the interface and the I/O process.

The following ISR3 bits indicate the status of the FIFOS:

- NFF Not Full FIFO
- NEF Not Empty FIFO
- INTSRC2 TNT4882 Setup for FIFO at Least Half Full; TNT4882 Setup for FIFO at Least Half Empty

The following ISR3 bit indicates the status of the interface:

- TLCINT Either an error has occurred or End has been received

The following ISR3 bit indicates the status of the I/O process:

- DONE I/O Process is complete

Handling Noninterrupt Programmed I/O

With noninterrupt programmed I/O, while you read data you must check certain status bits to determine if new data has arrived. While you write data, you must check certain status bits to determine if space is available in the FIFOs.

Noninterrupt programmed I/O is the fastest and most efficient method of reading and writing data for small transfer sizes (generally < 500 bytes). For larger transfer sizes, DMA I/O is more efficient for reading and writing data.

Inputting Data

See the `GPIB_PROG_IO() & User_GPIB_IO()` source code listings in `ngpib_io.c`.

1. Determine that the TNT4882 is listen addressed by reading the ADSR and checking for the LA bit.

2. Set up the TNT4882 to input data. (See the *Setting Up the TNT4882 for I/O* section, which is located earlier in this chapter.)

3. Read the ISR3 and check for the Done (DONE) bit. If DONE is set, the input process has ended successfully. Go to step 7.

4. (If using timeouts) Check ISR0 for the TO bit. If the TO bit is set and the FIFOs are empty (DEF not set), go to step 7.

5. Compare ISR3 to three cases:

   - (NEF|INTSRC2) or (NEF) — NFF is not set, so the FIFO is full. 15 data words or 30 bytes can be read from the FIFOs.

   - (NEF|INTSRC2|NFF) — NEF and INTSRC2 are set, so the FIFO is at least half full. NFF is set, so the FIFO is not completely full. Read only 8 words or 16 bytes from the FIFOs.

   - (NEF|NFF) — INTSRC2 is not set and NEF is set, so at least one word or byte is still in the FIFO. Read one word from the FIFOs.

6. Return to step 3.

7. Clean up I/O. (See the *Cleaning Up I/O* section, which is located later in this chapter.)

<u>Outputting Data</u>

See the `GPIB_PROG_IO() & User_GPIB_IO()` source code listings in `ngpib_io.c`.

1. Determine that the TNT4882 is talk addressed by reading the ADSR and checking for the TA bit.

2. Set up the TNT4882 to output data. (See the *Setting Up the TNT4882 for I/O* section, which is located earlier in this chapter.)

3. Read the ISR3 and check for the DONE and TLCINT bits. If neither bit is set, go to step 4. If DONE is set, the output process has ended successfully: go to step 6. If TLCINT is set, a transfer error has probably occurred: go to step 6.

4. Compare ISR3 to three cases:

• (NFF|INTSRC2) or (NFF)	NEF is not set, so the FIFO is empty. 16 data words or 32 bytes can be written to the FIFOs.
• (NEF|INTSRC2|NFF)	NFF and INTSRC2 are set, so the FIFO is at least half empty. NEF is set, so the FIFO is not completely empty. Write only 8 words or 16 bytes to the FIFOs.
• (NEF|NFF)	INTSRC2 is not set and NFF is set, so at least one word or space is left in the FIFO. Write one word to the FIFOs.

5. Return to step 2.

6. Clean up I/O. (See the *Cleaning Up I/O* section, which is located later in this chapter.)

Handling Interrupt-Driven Programmed I/O

In structure, interrupt-driven programmed I/O is almost identical to noninterrupt-driven programmed I/O. In interrupt-driven programmed I/O, the input and output cases are broken into two routines: `Read_Word()` and `Write_Word()`. These routines are identical to the two cases in noninterrupt-driven programmed I/O, but in the interrupt package, the interrupt for the Not Empty FIFO (NEF) function calls the function for inputting data, `Read_Word()`, and the interrupt for the Not Full FIFO (NFF) function calls the function for outputting data, `Write_Word()`. `DONE_Interrupt_ Handler()`, which is called on either DONE or TLCINT, handles the task for cleaning up I/O. For more information about `DONE_Interrupt_Handler()`, see the *Cleaning Up I/O* section, which is located later in this chapter.

Inputting Data

See the `GPIB_PROG_IO()` and `Read_Word` source code listing in `gpib_io.c`.

1. Determine that the TNT4882 is listen addressed by reading the ADSR and checking for the LA bit.

2. Set up the TNT4882 to input data. (See the *Setting Up the TNT4882 for I/O* section, which is located earlier in this chapter.)

3. Set a *Read* interrupt routine, like the `Read_Word()` function, to occur on the NEF interrupt status bit. The read function is the same in interrupt-driven programmed I/O and noninterrupt-driven programmed I/O, except that when the FIFOs become empty (~NEF) in interrupt-driven programmed I/O, the routine exits and waits to be called again by another NEF interrupt status bit assertion. (See the *Inputting Data* subsection under the *Handling Noninterrupt Programmed I/O* section, which is located earlier in this chapter.)

4. Set a *Clean Up* interrupt routine, like `DONE_Interrupt_Handler()`, to occur on interrupt status bits DONE and TLCINT. (See the *Cleaning Up I/O* section, which is located later in this chapter.)

Outputting Data

See the `GPIB_PROG_IO()` and `Write_Word()` source code listing in `gpib_io.c`.

1. Determine that the TNT4882 is talk addressed by reading the ADSR and checking for the TA bit.

2. Set up the TNT4882 to output data. (See the *Set Up the TNT4882 to Output Data* section, which is located earlier in this chapter.)

3. Set a *Write* interrupt routine, like the `Write_Word()` function, to occur on the NFF interrupt status bit. The write function is the same in interrupt-driven programmed I/O and noninterrupt-driven programmed I/O, except that when the FIFOs become full (~NFF) in interrupt-driven programmed I/O, the routine stops and waits to be called again by another NFF interrupt status bit assertion. (See the *Outputting Data* subsection under the *Handling Noninterrupt Programmed I/O* section, which is located earlier in this chapter.)

4. Set a *Clean Up* interrupt routine, like `DONE_Interrupt_Handler()`, to occur on the interrupt status bits DONE and TLCINT. (See the *Cleaning Up I/O* section in this chapter.)

Interrupt and Noninterrupt DMA I/O: GPIB_DMA_IO()

Using DMA can greatly increase the throughput of the TNT4882 interface for large transfer sizes. With DMA, an independent DMA Controller reads or writes each byte from or to the FIFOs. In programmed I/O, each byte from or to the FIFOs is read or written manually.

After the DMA Controller is programmed with an address, a count, and a direction of data transfer, it can be accessed by the TNT4882 to write data to or read data from memory. The DMA Controller updates its address pointer and decrements its count with every transfer.

The ESP code for programming the DMA Controller is written for the Intel 8237 Controller that is present in AT-style machines. Memory on the AT machines is arranged in 64-KB segments; each segment is called a *page*. In each segment, up to 64 KB can be accessed; a location inside the segment is called the *offset*. If it has been programmed with the page and offset of the location, the DMA Controller can access a specific memory location. In the case of the AT machine, the DMA Controller must be reprogrammed at each page boundary (64-KB segments), because the Address Page Register does not increment when the offset into the page rolls over.

Handling Noninterrupt DMA I/O with the Intel 8237 DMA Controller

Complete the steps in the section below to program the TNT4882 with the 8237 DMA Controller.

Inputting and Outputting Data

See the `GPIB_DMA_IO() & User_GPIB_IO()` source code listings in `ngpib_io.c`.

1. Determine that the TNT4882 is talk or listen addressed by reading the ADSR and checking for the TA or LA bit.

2. The DMA Controller can perform 16-bit and 8-bit accesses. The AT evaluation board, however, uses only 16-bit DMA channels. Therefore, if the start address of the data buffer that must be transferred is odd, the first byte of the I/O must be performed by using programmed I/O. (See the *Interrupt and Noninterrupt Programmed I/O* section, which is located earlier in this chapter.)

3. Convert the address of the buffer that must be accessed from a virtual to a physical offset and page. Both Microsoft C and Borland C provide the `FP_SEG()` and `FP_OFFSET()` functions to do this.

4. Calculate the number of bytes until a 64-KB segment boundary is reached.

5. If the transfer byte count requested is larger than the bytes to the boundary, a segment will be crossed. The TNT4882 and DMA Controller should be programmed with the smaller of the two counts if you are using the 8237. The DMA Controller will have to be reprogrammed for another transfer, because the 8237 does not increment its page register.

6. Set up the TNT4882 for I/O. (See the *Setting Up the TNT4882 for I/O* section in this chapter.)

7. Set the channel mask bit in the DMA Controller to disable the DMA channel that the TNT4882 is using.

8. Write the data offset and page address to the address registers of the DMA Controller.

9. Write the word count of the number of bytes to be transferred to the DMA register of the DMA Controller.

10. Write the I/O operation type (input or output) to the *mode* register of the DMA Controller.

11. Clear the Channel Command Register to initiate a transfer.

12. Turn on the DMA channel of the TNT4882 by clearing the Channel Mask bit of the DMA Controller.

13. Wait for DONE or TLCINT. If DONE is set, I/O has completed successfully.

    (INPUT operation) TLCINT means an END was received.

    (OUTPUT operation) TLCINT means an error was received.

14. Clean up I/O. (See the *Cleaning Up I/O* section in this chapter.)

15. (OUTPUT operation) If all data was not written to the GPIB bus, update the buffer pointer and reprogram the DMA Controller for another write: go to step 1.

    (INPUT operation) If all data was not read from the bus, END was not received. Update the buffer pointer and reprogram the DMA Controller for another read: go to step 1.

Handling Interrupt-Driven DMA I/O

Interrupt-driven DMA is slightly misleading, because only the clean-up process and, if necessary, the reprogramming of the Controller, are interrupt driven. The interrupt case of DMA is identical to the noninterrupt case, with two exceptions:

1. For an odd-starting buffer address, an interrupt-driven programmed I/O routine should handle the transfer of the first byte.

   (See step 1 under the *Inputting and Outputting* subsection of the *Handling Noninterrupt DMA I/O with the Intel 8237 DMA Controller* section in this chapter.)

2. Do not wait for DONE or TLCINT. Set a clean-up interrupt routine, like `DONE_Interrupt_Handler()`, to occur on interrupt status bits DONE and TLCINT. This clean-up interrupt routine should be able to reprogram the DMA Controller if another transfer is needed. (See the *Cleaning Up I/O* section in this chapter.)

(See steps 12, 13, and 14 under the *Inputting and Outputting* subsection of the *Handling Noninterrupt DMA I/O with the Intel 8237 DMA Controller* section in this chapter.)

Cleaning Up I/O: DONE_Handler() & DONE_Interrupt_Handler()

After an I/O transfer has completed either because it has properly terminated or an error condition has occurred, you must perform a few operations, such as clearing interrupts or errors, reading the transfer count, and possibly reinitiating another transfer, before you implement more I/O functions.

Noninterrupt and Interrupt Clean Up

See the `DONE_Handler()` source code listing in `ngpib_io.c` and see the `DONE_Interrupt_Handler()` source code in `gpib_io.c`.

1. Stop the FIFOs by writing the Stop command to the CMDR.

2. Write the Reset FIFO command to the CMDR to reset the FIFOs.

3. (If using DMA) Turn off the Intel 8237 DMA Controller channel that the TNT4882 is using.

4. (If using DMA) Turn off DMA on the AT board.

5. (If using DMA) Clear the channel status register of the DMA Controller.

6. If the END bit in ISR1 is set, clear the interrupt bit by writing the auxiliary command `0x55` (Clear END) to the AUXMR.

7. If the Error (ERR) bit in ISR1 is set, an error occurred during the transfer. To clear this bit, write the auxiliary command `0x57` (Clear ERR) to the AUXMR. If the TNT4882 was outputting or *sourcing* data and an error occurred, toggle the GO2SIDS bit in the HSSEL. This action forces the TNT4882 into a Source Idle State (SIDS). Write the type of handshake mode back to the HSSEL.

8. (If using timeouts) Check the TO bit in ISR0. Disable the TIMER by writing `0x00` to the AUXRJ.

9. Retrieve the I/O count by reading the CNT0, CNT1, CNT2, and CNT3 I/O count registers.

   count = original_cnt + (cnt3<<24)|(cnt2<<16)|(cnt1<<8)|(cnt0).

10. (Interrupts Only) Prevent interrupt functions from occurring again by disabling their mask bits:

    - Clear IMR3 for interrupts on DONE and TLCINT.
    - Clear IMR3 for NFF, if you are performing programmed I/O writes.
    - Clear IMR3 for NEF, if you are performing programmed I/O reads.

11. (Interrupts Only) If I/O has not completed (as in the case of crossing a segment boundary in DMA reads and writes), update the buffer pointer by the current transfer count and initiate another transfer.

Note: ***It is possible when doing word reads to overwrite a byte at the end of a buffer. The following is an example of this process:***

If the start I/O address is even and an odd byte read is performed, the last byte in the buffer will be overwritten by the WORD reads if the exact requested transfer count is reached. If the overwritten byte is important, you can correct this problem several ways:

1. ***Allocate a buffer large enough to hold data.***
2. ***Do a word read of the count-1, then set up the TNT4882 to perform a single-byte read. (This is how the ESP solves this problem.)***
3. ***Save the byte at the end of the buffer before doing a read, then replace the byte at the end of the transfer.***

Optimizing I/O

Some alterations can be made to the existing ESP source code to help improve the throughput. These changes are not implemented to make the source code easier to read.

1. Test I/O when you use DMA.
2. Eliminate loops.
3. Eliminate unnecessary tasks.
4. Do not use interrupts.
5. Compile: check the assembly code.

I/O Testing

For every system, there is a point where programmed I/O is faster than DMA, and a point where DMA is faster than programmed I/O. Programmed I/O is usually faster for small transfer sizes (transfer sizes under 500 bytes). You can easily find the crossover point: use DMA and programmed I/O to transfer progressively larger byte sizes, then compare the transfer times. Fortunately, the crossover point is generally about the same number of bytes whether the TNT4882 is performing reads or writes. You can then compare this crossover point to the desired transfer count and use it as a switch to decide when to use programmed I/O or DMA. Add this optimization to the `User_GPIB_IO()` (in `ngpib_io.c` and `gpib_io.c`) and `Queue_Interrupt_Handler()` functions. For the interrupt ESP, make sure you also change the global `Current_IO_Info.method` to reflect the change.

Eliminate Loops

You can speed up the programmed I/O routines in `gpib_io.c` and `ngpib_io.c` by removing the *for* loops and replacing them with the number of sequential reads or writes. This action can improve throughput about 5% for normal transfer sizes; throughput improvement can be greater than 5% for large transfer sizes. Eliminating these loops not only increases the code size but also significantly increases the throughput. Make this change in `PROG_GPIB_IO()` for the noninterrupt ESP and in `Write_Word()` and `Read_Word()` for the interrupt-driven ESP.

Eliminate Unnecessary Tasks

`Setup_TNT_IO()`

The `Setup_TNT_IO()` function performs a few operations that you can delete if you do not use the following functionality.

- Eliminate the timeout code if you are not using timeouts.

`DONE_Interrupt_Handler()` and `DONE_Handler()`

The DONE handlers perform a few operations that you can delete or skip for special cases.

1. Eliminate the timeout code if you are not using timeouts.
2. Eliminate the multiple address code if you are not using multiple addresses.

3. The `Get_DATA_COUNT()` function reads several registers, shifts their contents, and adds them together. Being clever about when to execute this function can save time.

   - (When outputting data) If no error occurs at the end of the write, you do not need to execute `Get_DATA_COUNT()`. The amount of data transferred is the same as the initial data count, global value `Requested_Count` (saved by the `Setup_TNT_Routine()`).

   - (When inputting data) If no error occurs and the END bit is not set, you do not need to execute `Get_DATA_COUNT()`. The amount of data transferred is the same as the global value `Requested_Count` (saved by the `Setup_TNT_Routine()`). If an END bit is set, execute `Get_DATA_COUNT()` to find out how many bytes were received.

   - If an error occurs in the I/O transfer, the count may not be important to the device program. The count may be skipped.

`Interrupt_Handler()`

The main interrupt handler is written to be generic and dynamic enough to handle any interrupt case that is registered by a `Set_Interrupt_Function()` call. You can eliminate the code that makes the `Interrupt_Handler()` flexible and rewrite it to handle only specific cases that the device will execute. To reduce the overhead of the `Interrupt_Handler()`, you can complete the following steps:

1. Eliminate the `Update_INTERFACE_STATUS()` function. Read only the registers that are necessary to handle the interrupts that have been set.

2. Replace the internal structure of the `Interrupt_Handler()` with a *case* statement for each interrupt function that might be executed. This action eliminates the need for the routine `Call_Interrupt_Function()`. The main interrupts that should be covered by this case statement are DONE, NEF, NFF, TLCINT, DET, DEC, and possibly IFC.

Do Not Use Interrupts

The interrupt routines let asynchronous processes occur and be handled transparently to the main routine. However, this process produces a small I/O time penalty, making it slower than a noninterrupt-driven I/O call. The overhead of jumping into the `Interrupt_Handler()` and checking the interrupt status registers slows I/O around 5 to 10%. If queues are not desired and there is no need for asynchronous I/O calls, use the noninterrupt I/O functions in `ngpib_io.c.`

Compiling: Check the Assembled Code

When you are compiling, always use the option for compiling for the fastest code (-O2 and /Ot for Borland and Microsoft, respectively). Most compilers also provide an option to generate assembled code for viewing. You should look at the assembled code to check for loops and operations that happen repeatedly, then you should try to eliminate unnecessary operations or register writes.

Interrupts and the TNT4882 ESP-488TL (inter_io.c)

The TNT4882 ESP-488TL interrupt source code is written for the Intel 8259 Programmable Interrupt Controller. The `Enable_GPIB_Interrupts()` function (located in `Initialize_Interface()`) loads the vector for the hardware interrupt channel defined in the `io_param.h` configuration file (`INTERRUPT_CHANNEL`). The function that this vector points to is `Interrupt_Handler()`.

Enabling TNT4882 Interrupts

The interrupts for the ISR0, ISR1, and ISR2 of the TNT4882 have a hierarchical design. Interrupts from ISR0, ISR1, and ISR2 must pass through ISR3 to get to the interrupt channel. Interrupts from ISR0, ISR1, and ISR2 are channeled through TLCINT in ISR3, so the TLCINT mask bit must be set when you attempt to get an interrupt from one of these registers. When you detect an interrupt from one of these registers (ISR0, ISR1, or ISR2), the TLCINT is the condition to interrupt on. A Global Interrupt Enable (GLINT) bit in IMR0 must be set to let interrupts pass to ISR3. You should keep a memory copy of the mask that is set for each interrupt mask register, because when an interrupt is received, you need to detect which interrupts have been enabled.

Initially Enabling Interrupt Handler

See the `GPIB_Interrupt_Control()` source code in `inter_io.c`.

1. Read in the interrupt mask register of the 8259.

2. Turn on the interrupts of the AT development board by writing `0x01` to the Interrupt Register (INTR).

3. Use the `_setvect()` function to set the address of the `Interrupt_Handler()` for the interrupt vector.

4. Enable the interrupts of the TNT4882 to occur by clearing the channel mask bit of the interrupt mask register of the 8259.

Enabling TNT4882 Interrupts For IMR0, IMR1, and IMR2

1. Make sure that the GLINT bit in IMR0 is set.

2. Write the desired mask to the mask register(s) (IMR0, IMR1, and IMR2).

3. Set the TLCINT bit in IMR3.

4. When the masked condition occurs, the TLCINT bit sets along with the INT bit in ISR3.

Enabling an ISR3 Interrupt to Occur

1. Write the desired mask to IMR3. The interrupt is set when the ISR3 condition arises.

Detection of the Interrupt Condition

The Interrupt Handler is called when a set interrupt condition arises. The Interrupt Handler should check the set masks against the conditions that arise, and it should call the proper routine. The interrupt function registered with this interrupt bit must be able to handle this condition and clear its interrupt status bit.

Detecting an Interrupt Condition

1. Read the ISR3. If the INT bit is not set, the interrupt came from somewhere else and you should exit the handler. If the INT bit is set, go to step 2.

2. If the TLCINT bit is not set in ISR3, you do not have to read ISR0, ISR1, ISR2, because no interrupt has occurred from these registers. If the TLCINT bit is set, read the registers.

3. Determine which interrupt routines to call: compare the memory copies of the interrupt masks to the read interrupt status register values and call the routines. The routines should clear their interrupt status bits.

4. Go to step 1.

Clearing an Interrupt Condition

There are three methods of clearing interrupts, and these methods depend on whether the SISB bit is set in AUXRI. If the SISB bit is not set, interrupts are cleared by reading the ISRX registers, and you must maintain a memory copy of the registers. If the SISB bit is set, an interrupt condition can be cleared by either issuing an auxiliary command like clrDEC (clear Device Clear status bit) or by performing an appropriate action. It is easiest to handle interrupt conditions when SISB is set.

<u>Three Methods of Clearing Interrupt Conditions</u>

1. If the SISB BIT is cleared, read ISR0, ISR1, and ISR2 and keep memory copies.
2. If the SISB bit is set, write a `clrXXX` command to the AUXMR to clear a status bit.
3. If the SISB bit is set or cleared, perform an action to clear it (for example, read FIFO to clear the NEF Interrupt Status bit).

How the Interrupt_Handler() Calls an Interrupt Routine

The `Interrupt_Handler()` is called when a TNT4882 interrupt occurs. `Set_Interrupt_Function()` should associate a function with the interrupt conditions. `Set_Interrupt_Function()` performs three important operations:

- Sets the appropriate interrupt mask bit.
- Updates a copy of this mask in a global mask variable of the form `MR_imrX.`
- Places a pointer to the registered function into an array of function pointers for the interrupt mask register (`MR_imrX`).

The routine that `Set_Interrupt_Function()` registers should be able to clear its own interrupt status bit. When the interrupt condition occurs, the `Interrupt_Handler()` routine reads the ISR3 and checks the INT bit. If the INT bit is set, the interrupt handler reads the status registers (ISR2, ISR1, and ISR0). The `Interrupt_Handler()` compares the global mask values (`MR_imrX`) to the interrupt status registers. If a condition in `MR_imrX` matches a status bit, the routine corresponding to that bit is called.

Example:

1. Register an interrupt on Device Clear.

`void Device_Clear(void)`	Declare Device Clear function.
`{`	
`TNT_Out(R_auxmr, F_clrDEC);`	Clear the status bit.
	Do something.

```
}
void main(void)
{
Initialize_Interface();          Enable the interface and set up
the interrupt handler.
Set_Interrupt_Function(R_imr1, B_dec, Device_Clear);
/* on 'dec' call Device_Clear */
..........
}
```

Event Handling

The device program should be able to handle certain events that require specific actions according to the IEEE 488.2 standard. The `INTERFACE_STATUS` status word contains a few bits that must be cleared manually if one of the DCAS, DTAS, IFC, SPOLL, or ERR conditions arises and is used by the device program. These event-handling routines can be called by interrupt routines in the case of DCAS, DTAS, IFC, and SPOLL by setting an interrupt on these bits with the routine `Set_Interrupt_Function()`.

Clearing the INTERFACE_STATUS Bits

ERR — Can be cleared programmatically by an error handler (`INTERFACE_STATUS &=~ERR`).

IFC — Can be cleared programmatically by a clrIFC auxiliary command. The interrupt ESP package provides a routine to clear the IFC condition. The `IFC_Interrupt_Handler()` in `gpib_io.c` can be enabled by the `io_param.h` configuration setting `USE_EVENT_HANDLERS`. `IFC_Int_Handler()` is not complete and should be changed for your device.

SPOLL — Can be cleared programmatically by writing a new STB to the SPMR (you can use `Set_4882_Status()`). This condition arises only if the Status Byte Out (STBO) bit in IMR0 is set. The interrupt ESP package provides a simple routine to handle SPOLL. You should change this routine for your device if you want to use it. The noninterrupt ESP contains a configuration variable, `USE_SPOOL_BIT`, that stops the sourcing of the serial poll byte until a new byte is written to the SPMR.

DTAS Should be cleared programmatically by a clrDET auxiliary command. The interrupt-driven ESP package provides a routine to handle the DTAS condition (`DET_Interrupt_Handler()`). The `DET_Interrupt_ Handler()` in `gpib_io.c` can be enabled by the `io_param.h` configuration setting `USE_ EVENT_HANDLERS`. If the queues are enabled, the DTAS message is placed in the Input Queue's current message terminator. `DET_Interrupt_Handler()` is not complete because it clears only the DTAS message. Thus, a routine should be added to actually implement the device trigger as requested by the IEEE 488.2 standard.

DCAS Can be cleared programmatically by a clrDEC auxiliary command. The interrupt-driven ESP package provides a routine to handle the DCAS condition (`DEC_Interrupt_Handler()`). The `DEC_Interrupt_ Handler()` in `gpib_io.c` can be enabled by the `io_param.h` configuration setting `USE_ EVENT_HANDLERS`. If the queues are enabled, the DCAS message is placed in the Input Queue's current message terminator. `DEC_Interrupt_Handler()` is not complete because it clears only the DCAS message. Thus, a routine should be added to actually implement the device clear as requested by the IEEE 488.2 standard.

Including Events in the Input Queue and Creating New Interrupt Event Handlers

To enable Device Clear and Device Trigger events to be registered in the queues, set `INCLUDE_EVENT_HANDLERS` in the `io_param.h` configuration file. This action lets the `DET_Interrupt_Handler()` and `DEC_Interrupt_Handler()` functions in `Initialize_Interface()` (`gpib_io.c`) be loaded.

Example: Creating a New Event IFC for the Input Queue.

1. Use the `DET` or `DEC_Interrupt_Handler()` as a model for creating a new routine.

```
void IFC_Interrupt_Handler(void)
{
int save_io_type;
TNT_Out(R_auxmr,F_clrIFC);       Clear the IFC Interrupt.
                                 Insert an IFC message in the   */
#if(USE_QUEUES)                  If there is an Input Queue.
 save_io_type = Current_IO_Info.io_type;
                                 Save the I/O type.
 Current_IO_Info.io_type = EVENT;
                                 Set to EVENT.
 Update_Queue_Info();            Place EVENT in the queue.
```

```
 INTERFACE_STATUS R = ~IFC;
 Current_IO_Info.io_type = save_io_type;
                                        Replace the I/O type.
#endif
}
```

2. Edit `Update_Queue_Info()` (`queue_io.c`) and update the terminator message in the Input Queue to include IFC.
   ```
   Input_Queue_Info.message[Input_In_Msg_Index].term| =
              INTERFACE_STATUS&(EOS|END|DTAS|DCAS|IFC);
   ```

Multichip Communications

The TNT4882 package is written to access only one chip. If a multidriver must control more than one chip from a single processor, the best way to convert the `gpib_io.c` or `ngpib_io.c` is at the `TNT_Out()` and `TNT_In()` functions listed in `gpib_io.h` and `ngpib_io.h`.

1. Create as many `gpib_io.h`, `inter_io.h`, `queue_io.h`, and `ngpib_io.h` header files as chips. Redefine the functions that are needed to reflect the chip that will be accessed.

   Example:
   ```
   chip1 [for gpib_io.h1]  #define Send1(buf,cnt,term)
         Send(buf,cnt,term)
   chip2 [for gpib_io.h2]  #define Send2(buf,cnt,term)
         Send(buf,cnt,term)
   ..... ..........
   chipn [for gpib_io.hn]  #define Sendn(buf,cnt,term)
         Send(buf,cnt,term)
   ```

2. Create as many `io_param.h` files as there are chips to access. Change the `TNT_BASE_IO` address `INTERRUPT_CHANNEL`, `DMA_CHANNEL`, and so on, as needed.

   Example:
   `io_param.h1`, `io_param.h2`, . . . `io_param.hn`

3. Change the include files in `gpib_io.c`, `inter_io.c`, `queue_io.c`, or `ngpib_io.c` for each new module, and compile `gpib_io.c`, `queue_io.c`, and `inter_io.c` into their object modules for every chip. Make these new object file names distinct, or the names will overwrite each other.

4. Write a device program.

```
#include gpib_io.h1
#include gpib_io.h2
  .........
#include gpib_io.hn

#include inter_io.h1
#include inter_io.h2
  .........
#include inter_io.hn

void main(void)
{
   Initialize_Interface1();
   Initialize_Interface2();
      ..........
   Initialize_Interfacen();
      ..........
   Send1(....);
      ..........
   Receive2(...);
}
```

6. Compile the `main()` program and link it with the new object modules.

IEEE 488.2 Status Model

The IEEE 488.2 standard status reporting model has been constructed in the `Update_4882_Status()` function. The IEEE 488.2 status reporting model has four required registers. (Refer to the IEEE 488.2 standard.)

Required IEEE 488.2 Status Registers

STB Status Byte Register (also called the Serial Poll Byte)

STB bit definitions:

x	RQS	ESB	MAV	x	x	x	x

- RQS Requesting Service bit. The device is asserting the service request (SRQ).
- ESB Event Status bit. A device event that the Controller has enabled in the ESE has occurred.
- MAV Message Available bit. Data is waiting to be read from the device.
- x don't care bits. These bits are defined by the device designers.

SRE Service Request Enable register. SRE compares its mask to the STB if the mask matches any bits in the STB. The RQS bit in STB is set and the GPIB shows an SRQ signal.

ESR Event Status Register. ESR is defined by the IEEE 488.2 standard in order to reflect various states of the device, such as power on and device error.

ESE Event Status Enable register. ESE compares its mask to ESR. If the mask matches any bits in ESR, the Event Status (ESB) bit will be set in STB.

Adding New Registers to the Status Model

You can add other registers to the required IEEE 488.2 set by editing the `Update_4882_Status()` function and the header file for the GPIB source code (`ngpib_io.c` or `gpib_io.c`).

Example:
Add two registers, Device-Defined Conditions (DDC) and Device Conditions Enable (DCE), and a new STB bit, DCB, in bit position 2 (the Device Condition bit). The bits in DDC can be used to describe any information that you would like to convey. Adding DCB to STB can set the service request line if a device condition in DDC is set.

1. Add to the register list (`MR_4882_Status[ ]`) in `gpib_io.h` or `ngpib_io.h`.

```
#define DDC 5          Device conditions register.
#define DCE 6          Device conditions enable the DCB register.
#define DCB (1<<2)     Device conditions bit.
```

2. Increase the array space of `MR_4882_Status[ ]` in `gpib_io.c` or `ngpib_io.h`.

3. Add the comparison to DDC and DCE to update the STB byte before STB and SRE are compared in `Update_4882_Status()`.

```
MR_4882_Status[STB] |=
(MR_4882_Status[DDC]&MR_4882_Status[DCE]) ? DCB : 0;
```

4. To enable an SRQ on DCB.

```
Set_4882_Status(SRE, DCB);      Set SRQ on DCB.

Set_4882_Status(DCE, 0xff);     Set DCB on any DDC condition.
..........
Set_4882_Status(DDC, 0x8); write to DCB
                                DCB will set, then SRQ will set.
```

ESP Coding Conventions

The following is a description of the function and variable naming conventions that have been used to write the ESP code. You may find this information helpful when you review the source code.

Helpful Source Code Rules

1. All constants, locals, and global values have meaningful names, with the exception of loop counters.

2. Initials and abbreviations are capitalized.

3. All constants that are used by a user program are capitalized (for example, `DCAS`, `TACS`, `EOI`).

4. All constants that are used by low-level functions capitalize the first word, but the remaining words are lower case and not separated (for example, `F_reset`, `MR_isr0`, `B_to`).

5. All status globals that are used by a device program are capitalized (for example, `INTERFACE_STATUS`, `INTERFACE_ERROR`, `DATA_COUNT`).

6. All globals that are used by low-level functions capitalize the first letter in every word, but the remaining letters are lower case (for example, `Write_EOS_Byte`, `Read_EOS_Byte`, `Primary_Address`).

7. All function names capitalize the first letter in every word (for example, `Send()`, `Set_Interrupt_Function()`).

8. All local variables are lower case (for example, `count`, `bytes_to_boundary`).

9. For the TNT4882 register descriptions:

   `R_xxx` — is a Hardware Register.
   `MR_xxx` — is a Memory copy of the xxx Register.
   `HR_xxx` — is a Hidden Register.
   `B_xxx` — is a register Bit.
   `F_xxx` — is a Field of many bits.

Appendix A
Configuring the TNT4882 Talker/Listener ESP

This appendix explains how to configure the interrupt and noninterrupt ESP.

Configuring the Noninterrupt and Interrupt ESP

The interrupt and noninterrupt ESP contain an `io_param.h` configuration file. Use the `io_param.h` configuration file to set important hardware and software configuration options, such as the hardware I/O address of the TNT4882 board, the DMA channel, the GPIB address, the compiler type, and the memory model type. If a parameter is an optional configuration, a YES or NO constant is supplied to include the optional code.

The tables below describe the options and the valid values for parameters; related values are grouped together.

Addressing Parameters

Addressing parameters list configuration options for the addressing modes of the TNT4882. Table A-1 describes the addressing parameters and summarizes their valid values.

Table A-1. Addressing Parameters

Parameter	Valid Values	Default	Description
`ADDRESS_MODE`	0 to 6	0	`ADDRESS_MODE` sets the initial addressing mode for the TNT4882. There are seven possible modes: • 0 single primary • 1 single primary–single secondary • 2 single primary–multiple secondary • 3 multiple primary • 4 talk-only mode • 5 listen-only mode • 6 no addressing
`PRIMARY_ADDRESS`	0 to 30	1	GPIB primary address of the TNT4882. `PRIMARY_ADDRESS` is used for address modes 0, 1, and 2.
`SECONDARY_ADDRESS`	0 to 30	0	GPIB secondary address of the TNT4882. `SECONDARY_ADDRESS` is used for address mode 1.
`MULTIPLE_ADDRESSES`	0 to 30	{0}	GPIB multiprimary and multisecondary addresses of the TNT4882. For example, {1, 2, 5}. `MULTIPLE_ADDRESSES` is used for address modes 2 and 3.
`NUMBER_OF_ADDRESSES`	0 to 31	0	Number of addresses in `MULTIPLE_ADDRESS`.

I/O Parameters

I/O parameters list configuration options for the ESP I/O functions. Table A-2 describes the I/O parameters and summarizes their valid values.

Table A-2. I/O Parameters

Parameter	Valid Values	Default	Description
TIMEOUT_ FACTOR_INDEX	0 to 15	13	TIMEOUT_FACTOR_INDEX sets the amount of time that the TNT4882 waits before it issues a timeout. This index corresponds to the time values in Table 3-1. Zero disables timeouts.
USE_BYTE_ TIMEOUTS	YES/N O	YES	USE_BYTE TIMEOUTS lets the timer be reset for each byte. If this parameter is set to NO, the timeout index sets the maximum amount of time an entire transfer can take.
READ_EOS_BYTE	0x00 to 0xff	0x00	End-of-String byte for reads.
WRITE_EOS_ BYTE	0x00 to 0xff	0x00	End-of-String byte for writes.
USE_EIGHT_BIT _EOS_COMPARE	YES/N O	NO	Compare EOS 7 or 8 bits.
USE_TRANSMIT_ EOI_WITH_EOS	YES/N O	YES	This parameter sends EOI with the EOS byte for writes.
USE_HIGH_ SPEED_T1	YES/N O	YES	USE_HIGH_SPEED_T1 enables a short T1 delay while performing three-wire handshaking.
HS_MODE	ONE_ CHIP	ONE_ CHIP	TNT4882 chip mode.
CABLE_LENGTH	0 to 15	15	Total length of cable in the system.

Miscellaneous Handlers

Miscellaneous handlers list configuration options for the activation of special event handlers. Table A-3 describes the miscellaneous handlers and summarizes their valid values.

Table A-3. Miscellaneous Handlers

Parameter	Valid Values	Default	Description
USE_EVENT_ HANDLERS	YES/NO	NO	(For the interrupt ESP only) USE_EVENT_HANDLERS enables the loading of special interrupt handlers for the following events: • IFC (IFC) Interface Clear • DET (DTAS) Device Trigger • DEC (DCAS) Device Clear You should edit these routines to meet the IEEE 488.2 specifications for your device.
USE_SPOLL_BIT	YES/NO	NO	(For the noninterrupt ESP only) USE_SPOLL_BIT lets the SPOLL bit set in INTERFACE_STATUS. A serial poll is held off until a new byte is written to the SPMR. (For the interrupt ESP only) USE_SPOLL_BIT enables the loading of the STBO interrupt handler. You should edit this routine to meet the IEEE 488.2 specifications for your device.

(continues)

Table A-3. Miscellaneous Handlers (Continued)

Parameter	Valid Values	Default	Description
USE_HS488_ HANDLER	YES/N O	NO	USE_HS488_HANDLER lets the HS488 CFE and CFGN command bytes be detected. (For the noninterrupt ESP only) The Validate_CF_Command() function called by CPT_Handler() reads these configuration bytes. (For the interrupt ESP only) USE_HS488_HANDLER loads the CPT_Interrupt_Handler() that calls Validate_CF_Command().

Hardware Parameters

Hardware parameters set the configuration for the evaluation board hardware. Table A-4 describes the TNT4882, base I/O address, DMA, and interrupt hardware parameters and summarizes their valid values.

Table A-4. TNT4882, DMA, and Interrupt Hardware Parameters

Parameter	Valid Values	Default	Description
TNT_BASE_ ADDRESS	0x020 to 0x3e0	0x2c0	Board hardware base I/O address.
DMA_CHANNEL	5, 6, 7	5	The DMA channel that the evaluation board is using.
USE_DMA	YES/NO	YES	USE_DMA enables the use of the ESP's DMA code.
USE_DEMAND_ MODE_DMA	YES/NO	YES	USE_DEMAND_MODE_DMA enables the use of data-burst DMA transfers. Otherwise, DMA transfers use single-cycle DMA (see Intel 8237 DMA Controller documentation).
INTERRUPT_ CHANNEL	3 to 7; 9 to 12; 14, 15	11	(For the interrupt ESP only) The interrupt channel the evaluation board is using.

Software Configuration

Software configuration parameters set the compiler type and the I/O buffer pointer type. Table A-5 describes the software configuration parameters and summarizes their valid values.

Table A-5. Software Configuration

Parameter	Valid Values	Default	Description
USE_HUGE_ BUFFERS	YES/N O	YES	Set USE_HUGE_BUFFERS to YES if buffers will cross 64 KB segment boundaries.
USE_MICROSOFT _C	YES/N O	NO	Set USE_MICROSOFT_C to YES if using Microsoft C instead of Borland C.

INTERRUPT ESP ONLY

Queue Model Parameters

Queue model parameters let you set the queue sizes and the maximum number of stored messages. Table A-6 describes the queue model parameters and summarizes their valid values.

Table A-6. Queue Model Parameters (For the Interrupt ESP Only)

Parameter	Valid Values	Default	Description
USE_QUEUES	YES/NO	NO	USE_QUEUES enables the use of asynchronous queues for I/O.
INPUT_QUEUE_ SIZE	0 to 0xffffffff	10000	Input Queue data space size.
OUTPUT_QUEUE_ SIZE	0 to 0xffffffff	10000	Output Queue data space size.
MESSAGE_ BUFFER_SIZE	0 to 0xffff	100	Maximum number of terminated messages that can be stored in an Input or Output queue.
USE_EOS_WITH_ QUEUES	YES/NO	NO	USE_EOS_WITH_QUEUES lets the Input and Output Queues use EOS bytes for termination.

Appendix B
Programming Examples

This appendix provides six programming examples that demonstrate various I/O models and how to use some of the ESP function calls. You can also find these examples in the interrupt and noninterrupt directories on the distribution diskette.

Basic Device Program Model

1. Initialize the TNT4882 (`Initialize_Interface()`).
2. Update the status word to detect the address state or events (`Update_INTERFACE_STATUS()`).
3. If addressed to listen (`INTERFACE_STATUS&LACS`):
   - Read data from the GPIB (`Receive()/Receive_ASYNC()`).
   - Check for I/O errors (ERR).
   - Parse data for commands and execute instructions.
4. If addressed to talk (`INTERFACE_STATUS&TACS`):
   - Write data to the GPIB if a message is ready for output (`Send()/Send_ASYNC()`).
   - Check for I/O errors (ERR).
5. Update IEEE 488.2 status registers STB and ESR (`Set_4882_Status()`, `Clear_4882_Status`).
6. If device will be stopped or shut down, turn off the TNT4882 (`Interface_Off()`).
7. Go to step 2.

Interrupt ESP Examples

```
/************************************************************************
**
 *
 *  [Interrupt ESP]              "NORMAL I/O AND NORMAL ADDRESSING"
 *
 *  EX1.C :  This example program can read up to 50 bytes when addressed
 *           to listen and will write the message
 *
 *                "Device 1 addressed to talk"
 *
 *           when addressed to talk. Upon reading data the MAV bit of
 *           STB will be set and the SRQ signal asserted. The program
 *           will terminate when 'q' is pressed. The EABO error will occur
 *           when the interface tries to read/write and the control
 *           program does not transmit/accept data.
 *
 *  io_param.h....(default settings)
 *

*************************************************************************
*/


#include "io_param.h"
#include "gpib_io.h"

void main(void)
{
 char buf[50];
 char keystroke;

 Initialize_Interface();                  /* Initialize TNT             */
 Set_Timeout(11,TRUE);                    /* Change timeout to 1 sec.   */

 Set_4882_Status(SRE,0x10);               /* Set SRQ on MAV             */

 Set_Address_Mode(0);                     /* Set to single primary mode  */
 Change_Primary_Address(1);               /* Set primary address        */

 while(keystroke!='q') {                  /* Loop until 'q' pressed     */

   Update_INTERFACE_STATUS();             /* Update status word         */

   if(INTERFACE_STATUS&LACS) {            /* If listen addressed        */

     Receive(buf,50,EOI);                 /* Read up to 50 bytes        */

     /************************************
      *
      *  Parse commands/data
      *
      ************************************/

     if(DATA_COUNT>0) {                   /* If bytes were received     */
       Set_4882_Status(STB,0x10);         /*   Set MAV bit in STB       */
                                          /*   Print to stdio           */
       printf("\nReceived %lu data bytes",DATA_COUNT);
     }
   }
   else if(INTERFACE_STATUS&TACS) {       /* If talk addressed          */

                                          /* Send status data           */
     Send("Device 1 addressed to talk",26,EOI);
```

```
     if(DATA_COUNT>0) {                     /* If data bytes sent print it */
       printf("\nSent     %lu data bytes",DATA_COUNT);
       Clear_4882_Status(STB,0x10);         /*  Clear MAV bit              */
     }
   }

   if(kbhit())                              /* Get keystroke               */
     keystroke=getch();

   if(INTERFACE_STATUS&ERR) {               /* If EABO report it           */
     printf("\nGPIB error has occurred (INTERFACE_ERROR =
%d)",INTERFACE_ERROR);
     INTERFACE_STATUS&=~ERR;
   }
 }

 Interface_Off();                           /* Disable interface           */
}


/***************************************************************************
**
 *
 *  [Interrupt ESP]            "NORMAL I/O AND MULTIPLE ADDRESSING"
 *
 *  EX2.C :  This example program can read up to 50 bytes when addressed
 *           to listen at addresses 1,2,3,4,5.  When addressed to talk,
 *           it will respond with a message of the form
 *
 *                "Talking from address X"
 *
 *           Upon reading data the MAV bit of
 *           STB will be set and the SRQ signal asserted. The program
 *           will terminate when 'q' is pressed. The EABO error will occur
 *           when the interface tries to read/write and the control
 *           program does not transmit/accept data.
 *
 *  io_param.h....(default settings)
 *

****************************************************************************
*/


#include "io_param.h"
#include "gpib_io.h"

void main(void)
{
 char buf[50];
 char keystroke;
 int ad_list[5]={1,2,3,4,5};

 Initialize_Interface();                    /* Initialize TNT              */
 Set_Timeout(11,TRUE);                      /* Change timeout to 1 sec.    */

 Set_4882_Status(SRE,0x10);                 /* Set SRQ on MAV              */

 Set_Address_Mode(3);                       /* Set to multiple primary mode*/
 Change_Multiple_Addresses(ad_list,5);  /* Set multiple primary addresses*/

 while(keystroke!='q') {                    /* Loop until 'q' pressed      */

   Update_INTERFACE_STATUS();               /* Update status word          */

   if(INTERFACE_STATUS&LACS) {              /* If listen addressed         */
```

```
     Receive(buf,50,EOI);                /* Read up to 50 bytes          */

     /***********************************
      *
      *  Parse commands/data
      *
      ***********************************/

     if(DATA_COUNT>0) {                  /* If bytes were received       */
       Set_4882_Status(STB,0x10);        /*   Set MAV bit in STB         */
                                         /*   Print to stdio             */
       printf("\nReceived %lu data bytes at address
0x%04x",DATA_COUNT,MULTI_ADDRESS);
     }
   }
   else if(INTERFACE_STATUS&TACS) {      /* If talk addressed            */

     MULTI_ADDRESS=CURRENT_ADDRESS;      /* Set address global           */

     switch(MULTI_ADDRESS) {
       case (0x0100):                    /* Write from address 1         */
         Send("Talking from address 1",22,EOI);
         break;

       case (0x0200):                    /* Write from address 2         */
         Send("Talking from address 2",22,EOI);
           break;

       case (0x0300):                    /* Write from address 3         */
         Send("Talking from address 3",22,EOI);
         break;

       case (0x0400):                    /* Write from address 4         */
         Send("Talking from address 4",22,EOI);
         break;

       case (0x0500):                    /* Write from address 5         */
         Send("Talking from address 5",22,EOI);
         break;
     }

     if(DATA_COUNT>0) {                  /* If data bytes sent print it */
       printf("\nSent     %lu data bytes from address 0x%04x",DATA_COUNT,
MULTI_ADDRESS);
       Clear_4882_Status(STB,0x10);      /*  Clear MAV bit               */
     }
   }

/*   printf("\nSTB=0x%02x",TNT_In(R_spmr));*/

   if(kbhit())                           /* Get keystroke                */
     keystroke=getch();

   if(INTERFACE_STATUS&ERR) {            /* If EABO report it            */
     printf("\nGPIB error has occurred (INTERFACE_ERROR =
%d)",INTERFACE_ERROR);
     INTERFACE_STATUS&=~ERR;
   }
 }

 Interface_Off();                        /* Disable interface            */
}


****************************************************************************
*/
 *
```

```
 *  [Interrupt ESP]        "SIMPLE QUEUE USAGE AND NORMAL ADDRESSING"
 *
 *  EX3.C :  This example uses the asynchronous I/O queues and can read up
 *           to 1000 bytes when addressed to listen at primary address 1.
 *           When addressed to talk, it will echo back the first and then
 *           the succeeding messages written to it.
 *
 *           Upon reading data from the input queue the MAV bit of
 *           STB will be set and the SRQ signal asserted. The program
 *           will terminate when 'q' is pressed.
 *
 *           Device Clear and Device Trigger events are place in the input
 *           queue.
 *
 *           The REM_Interrupt_Handler() is loaded for demonstration
 *           purpose only.
 *
 *  io_param.h....(default settings accept)
 *
 *  USE_EVENT_HANDLERS YES
 *
 *  USE_QUEUES YES
 *  INPUT_QUEUE_SIZE 1000
 *  OUTPUT_QUEUE_SIZE 1000
 *
***************************************************************************
*/


#include "io_param.h"
#include "gpib_io.h"
#include "queue_io.h"
#include "inter_io.h"

void REM_Interrupt_Handler(void);


void main(void)
{
 char buf[50];
 char keystroke;
 int term;

 Initialize_Interface();                  /* Initialize TNT            */

 Set_4882_Status(SRE,0x10);               /* Set SRQ on MAV            */

                                          /* Set an int function on B_rem*/
 Set_Interrupt_Function(R_imr2,B_remc,REM_Interrupt_Handler);

 Set_Address_Mode(0);                     /* Set to single primary mode  */
 Change_Primary_Address(1);               /* Set primary address       */

 while(keystroke!='q') {                  /* Loop until 'q' pressed    */

   if(Message_Available()) {              /* If message in input queue  */

                                       /* Read up to 1000 bytes from the */
                                          /*   input queue             */
     Read_Input_Queue(buf,Message_Length(),&term);

     /************************************
      *
      *  Parse commands/data
      *
      ************************************/
```

```
     if(term&DCAS)                        /* DCAS read from input queue  */
       printf("\nDevice Clear read from input queue");

     if(term&DTAS)                        /* DTAS read from input queue  */
       printf("\nDevice Trigger read from input queue");

     if(QUEUE_COUNT>0) {                  /* If bytes were read          */
       printf("\nInput Queue data written to Output Queue");
       Set_4882_Status(STB,0x10);         /* Set SRQ on MAV              */
       Write_Output_Queue(buf,QUEUE_COUNT,EOI);
     }

   }                                      /* If talk addressed           */

                                       /* If no data in the output queue */
   if(!(QUEUE_STATUS&OMWA) && (Read_4882_Status(STB)&0x10))
     Clear_4882_Status(STB,0x10);         /*   Clear MAV bit             */

   if(kbhit())                            /* Get keystroke               */
     keystroke=getch();

   if(INTERFACE_STATUS&ERR) {             /* If EABO report it           */
     printf("\nGPIB error has occurred (INTERFACE_ERROR =
%d)",INTERFACE_ERROR);
     INTERFACE_STATUS&=~ERR;
   }
 }

 Interface_Off();                         /* Disable interface           */
}

void REM_Interrupt_Handler(void)
{
  TNT_Out(R_auxmr,F_clrREMC);
  Set_4882_Status(STB,0x02);
}
```

```
/*****************************************************************************
**
 *
 *  [Interrupt ESP]     "QUEUES USAGE AND MULTIPLE ADDRESSING"
 *
 *  EX4.C :  This example uses the asynchronous I/O queues and can read up
 *           to 1000 bytes when addressed to listen at addresses 1,2,3,4,5.
 *           When addressed to talk, it will respond with a message of the
 *           form
 *
 *                "Talking from address X"
 *
 *           Upon reading data the MAV bit of
 *           STB will be set and the SRQ signal asserted. The program
 *           will terminate when 'q' is pressed. The EABO error will occur
 *           when the interface tries to read/write and the control
 *           program does not transmit/accept data.
 *
 *           An interrupt function has been included for REM. Upon entering
 *           REM state the function REM_Interrupt_Handler() will be called
 *           and a status bit in STB is set.
 *
 *  io_param.h....(default settings accept)
 *
 *  USE_QUEUES YES
 *  INPUT_QUEUE_SIZE 1000
 *

*****************************************************************************
*/


#include "io_param.h"
#include "gpib_io.h"
#include "queue_io.h"
#include "inter_io.h"

void REM_Interrupt_Handler(void);

void main(void)
{
 char buf[50];
 char keystroke;
 int ad_list[5]={1,2,3,4,5};
 int term;


 Initialize_Interface();                  /* Initialize TNT              */
                                          /* Set an int function on B_rem*/
 Set_Interrupt_Function(R_imr2,B_remc,REM_Interrupt_Handler);
 Disable_Output_Queue();                  /* Disable the output queue    */
 Set_Timeout(11,TRUE);                    /* Change timeout to 1 sec.    */

 Set_4882_Status(SRE,0x10);               /* Set SRQ on MAV              */

 Set_Address_Mode(3);                    /* Set to multiple primary mode */
 Change_Multiple_Addresses(ad_list,5);  /* Set multiple primary addresses*/

 while(keystroke!='q') {                  /* Loop until 'q' pressed      */

   Update_INTERFACE_STATUS();             /* Update status word          */

   if(Message_Available()) {              /* If message in input queue   */

                                        /* Read up to 1000 bytes from the*/
                                          /*   input queue               */
```

```
     Read_Input_Queue(buf,Message_Length(),&term);

     /************************************
      *
      *  Parse commands/data
      *
      ************************************/

     if(QUEUE_COUNT>0) {                  /* If bytes were read          */
       Set_4882_Status(STB,0x10);         /*   Set MAV bit in STB        */
                                          /*   Print to stdio            */
       printf("\nReceived %lu data bytes at address
0x%04x",QUEUE_COUNT,QUEUE_ADDRESS);
     }
   }                                      /* If talk addressed           */
                                          /*   and no ASYNC I/O          */
   else if((INTERFACE_STATUS&TACS) && (!(INTERFACE_STATUS&ASYNC))) {

     MULTI_ADDRESS=CURRENT_ADDRESS;       /* Set address global          */

     switch(MULTI_ADDRESS) {
       case (0x0100):                     /* Write from address 1        */
         Send_ASYNC("Talking from address 1",22,EOI);
         break;
       case (0x0200):                     /* Write from address 2        */
         Send_ASYNC("Talking from address 2",22,EOI);
         break;
       case (0x0300):                     /* Write from address 3        */
         Send_ASYNC("Talking from address 3",22,EOI);
         break;
       case (0x0400):                     /* Write from address 4        */
         Send_ASYNC("Talking from address 4",22,EOI);
         break;
       case (0x0500):                     /* Write from address 5        */
         Send_ASYNC("Talking from address 5",22,EOI);
         break;
     }

     if(DATA_COUNT>0) {                   /* If data bytes sent print it */
       printf("\nSent     %lu data bytes from address 0x%04x",DATA_COUNT,
MULTI_ADDRESS);
       Clear_4882_Status(STB,0x10);       /*  Clear MAV bit              */
     }
   }

   if(kbhit())                            /* Get keystroke               */
     keystroke=getch();

   if(INTERFACE_STATUS&ERR) {             /* If EABO report it           */
     printf("\nGPIB error has occurred (INTERFACE_ERROR =
%d)",INTERFACE_ERROR);
     INTERFACE_STATUS&=~ERR;
   }
 }

 Interface_Off();                         /* Disable interface           */
}

void REM_Interrupt_Handler(void)
{
  TNT_Out(R_auxmr,F_clrREMC);
  Set_4882_Status(STB,0x02);
}
```

Noninterrupt ESP Examples

```
/************************************************************************
**
 *
 *  [Non-Interrupt ESP]        "NORMAL I/O AND NORMAL ADDRESSING"
 *
 *  EX5.C :  This example program can read up to 50 bytes when addressed
 *           to listen and will write the message
 *
 *                "Device 1 addressed to talk"
 *
 *           when addressed to talk. Upon reading data the MAV bit of
 *           STB will be set and the SRQ signal asserted. The program
 *           will terminate when 'q' is pressed. The EABO error will occur
 *           when the interface tries to read/write and the control
 *           program does not transmit/accept data.
 *
 *           This program also uses the configuration option USE_SPOLL_BIT.
 *           With USE_SPOLL_BIT set to yes a new serial poll byte will
 *           not be sourced until the spmr is written to.
 *
 *  io_param.h....(default settings)
 *
 *  #define USE_SPOLL_BIT YES
 *

************************************************************************
*/


#include "io_param.h"
#include "ngpib_io.h"

void main(void)
{
 char buf[50];
 char keystroke;
 int spoll_count=0;

 Initialize_Interface();                /* Initialize TNT              */
 Set_Timeout(11,TRUE);                  /* Change timeout to 1 sec.    */

 Set_4882_Status(SRE,0x10);             /* Set SRQ on MAV              */

 Set_Address_Mode(0);                   /* Set to single primary mode  */
 Change_Primary_Address(1);             /* Set primary address         */

 while(keystroke!='q') {                /* Loop until 'q' pressed      */

   do{
     if(Read_GPIB_Lines()&0x8000)
       printf("\nATN asserted");

     Update_INTERFACE_STATUS();         /* Update status word          */

   } while(Read_GPIB_Lines()&0x8000);   /*  Continue when ATN not
asserted*/
                                        /*  Update_INTERFACE_STATUS()  */
                                        /*  calls CPT_Handler() which  */
                                      /*  validates the HS488 and mult-*/
                                    /*  primary addresses. These bytes*/
                                        /*  must be read from the GPIB */
                                     /*  before an I/O function can be */
                                        /*  called.                    */
```

```
   if(INTERFACE_STATUS&SPOLL) {           /* If SPOLL set              */
     Set_4882_Status(STB,spoll_count++);  /*   Set count of SPOLL      */
     Clear_4882_Status(STB,0xff);         /*   Clear STB               */
   }

   else if(INTERFACE_STATUS&LACS) {       /* If listen addressed       */

     Receive(buf,50,EOI);                 /* Read up to 50 bytes       */

     /**************************************
      *
      *  Parse commands/data
      *
      **************************************/

     if(DATA_COUNT>0) {                   /* If bytes were received    */
       Set_4882_Status(STB,0x10);         /*   Set MAV bit in STB      */
                                          /*   Print to stdio          */
       printf("\nReceived %lu data bytes",DATA_COUNT);
     }
   }
   else if(INTERFACE_STATUS&TACS) {       /* If talk addressed         */

                                          /* Send status data          */
     Send("Device 1 addressed to talk",26,EOI);

     if(DATA_COUNT>0) {                   /* If data bytes sent print it */
       printf("\nSent     %lu data bytes",DATA_COUNT);
       Clear_4882_Status(STB,0x10);       /*  Clear MAV bit            */
     }
   }

   if(kbhit())                            /* Get keystroke             */
     keystroke=getch();

   if(INTERFACE_STATUS&ERR) {             /* If EABO report it         */
     printf("\nGPIB error has occurred (INTERFACE_ERROR =
%d)",INTERFACE_ERROR);
     INTERFACE_STATUS&=~ERR;
   }
 }

 Interface_Off();                         /* Disable interface         */
}
```

```
/***********************************************************************
**
 *
 *  [Non-Interrupt ESP]       "NORMAL I/O AND MULTIPLE ADDRESSING"
 *
 *  EX6.C :  This example program can read up to 50 bytes when addressed
 *           to listen at addresses 1,2,3,4,5.  When addressed to talk,
 *           it will respond with a message of the form
 *
 *                "Talking from address X"
 *
 *           Upon reading data the MAV bit of
 *           STB will be set and the SRQ signal asserted. The program
 *           will terminate when 'q' is pressed. The EABO error will occur
 *           when the interface tries to read/write and the control
 *           program does not transmit/accept data.
 *
 *  io_param.h....(default settings)
 *

***********************************************************************
*/


#include "io_param.h"
#include "ngpib_io.h"

void main(void)
{
 char buf[50];
 char keystroke;
 int ad_list[5]={1,2,3,4,5};

 Initialize_Interface();                  /* Initialize TNT              */
 Set_Timeout(11,TRUE);                    /* Change timeout to 1 sec.    */

 Set_4882_Status(SRE,0x10);               /* Set SRQ on MAV              */

 Set_Address_Mode(3);                     /* Set to multiple primary mode*/
 Change_Multiple_Addresses(ad_list,5); /* Set multiple primary addresses */

 while(keystroke!='q') {                  /* Loop until 'q' pressed      */

   do{
     if(Read_GPIB_Lines()&0x8000)         /* If ATN asserted report it   */
       printf("\nATN asserted");

     Update_INTERFACE_STATUS();           /* Update status word          */
   } while(Read_GPIB_Lines()&0x8000);     /*  Continue when ATN not
asserted*/
                                          /*  Update_INTERFACE_STATUS()  */
                                          /*  calls CPT_Handler() which  */
                                       /*  validates the HS488 and mult- */
                                       /*  primary addresses. These bytes*/
                                          /*  must be read from the GPIB */
                                       /*  before an I/O function can be */
                                          /*  called.                   */

   if(INTERFACE_STATUS&LACS) {            /* If listen addressed         */

     Receive(buf,50,EOI);                 /* Read up to 50 bytes         */

     /************************************
      *
      *  Parse commands/data
      *
```

```
      ************************************/

     if(DATA_COUNT>0) {                  /* If bytes were received      */
       Set_4882_Status(STB,0x10);        /*   Set MAV bit in STB        */
                                         /*   Print to stdio            */
       printf("\nReceived %lu data bytes at address
0x%04x",DATA_COUNT,MULTI_ADDRESS);
     }
   }
   else if(INTERFACE_STATUS&TACS) {      /* If talk addressed           */

     MULTI_ADDRESS=CURRENT_ADDRESS;      /* Set address global          */

     switch(MULTI_ADDRESS) {
       case (0x0100):                    /* Write from address 1        */
         Send("Talking from address 1",22,EOI);
         break;
       case (0x0200):                    /* Write from address 2        */
         Send("Talking from address 2",22,EOI);
         break;
       case (0x0300):                    /* Write from address 3        */
         Send("Talking from address 3",22,EOI);
         break;
       case (0x0400):                    /* Write from address 4        */
         Send("Talking from address 4",22,EOI);
         break;
       case (0x0500):                    /* Write from address 5        */
         Send("Talking from address 5",22,EOI);
         break;
     }

     if(DATA_COUNT>0) {                  /* If data bytes sent print it */
       printf("\nSent     %lu data bytes from address 0x%04x",DATA_COUNT,
MULTI_ADDRESS);
       Clear_4882_Status(STB,0x10);      /*  Clear MAV bit              */
     }
   }

   if(kbhit())                           /* Get keystroke               */
     keystroke=getch();

   if(INTERFACE_STATUS&ERR) {            /* If EABO report it           */
     printf("\nGPIB error has occurred (INTERFACE_ERROR =
%d)",INTERFACE_ERROR);
     INTERFACE_STATUS&=~ERR;
   }
 }

 Interface_Off();                        /* Disable interface           */
}
```

Appendix C
Mnemonics Key

This appendix defines the mnemonics (abbreviations) that this manual uses for functions, remote messages, local messages, states, bits, registers, integrated circuits, and system functions.

The mnemonic types in the key are abbreviated to mean the following:

B	Bit
C	Command
F	Function
IM	Interface Message
LM	Local Message
Q	Queue
R	Register
RM	Remote Message
ST	State

Mnemonic	Type	Definition

A

Mnemonic	Type	Definition
ADMR	R	Address Mode Register
ADR	R	Address Register
ADSC	B	Address Status Change bit
ADSR	R	Address Status Register
APT	B	Address Pass Through bit
ARS	B	Address Register Select bit
ASYNC	B	Asynchronous I/O In Progress bit
ATN	RM	Attention
AUXCR	R	Auxiliary Command Register
AUXMR	R	Auxiliary Mode Register
AUXRA	R	Auxiliary Register A
AUXRB	R	Auxiliary Register B
AUXRE	R	Auxiliary Register E
AUXRF	R	Auxiliary Register F
AUXRG	R	Auxiliary Register G
AUXRI	R	Auxiliary Register I
AUXRJ	R	Auxiliary Register J

B

Mnemonic	Type	Definition
BTO	B	Byte Timeout bit

C

Mnemonic	Type	Definition
CCEN	B	Carry Cycle Enable bit
CFE		Configuration Enable
CFG	R	Configuration Register
CFGR	R	Configuration Register
clrDEC	B	Clear Device Clear Status Bit bit
clrDET	C	Clear Device Trigger Status Bit command
clrIFC	C	Clear Interface Clear Status Bit command
CMDR	R	Command Register
CNT[0-3]	B	Count Register bits 0 through 3
CNT0	R	Count 0 Register
CNT1	R	Count 1 Register
CNT2	R	Count 2 Register
CNT3	R	Count 3 Register
CPT	B	Command Pass Through bit
CPT ENAB	B	Command Pass Through Enable bit
CPTR	R	Command Pass Through Register

Mnemonic	Type	Definition

D

Mnemonic	Type	Definition
DAC		Data Accepted
DAV	RM	Data Valid
DCAS	B	Device Clear Active State bit
DCE		Device Conditions Enable
DCL	C	Device Clear multiline interface command
DDC		Device-Defined Conditions
DEC	B	Device Clear bit
DET	B	Device Trigger bit
DGA	B	Deglitch Selector A bit
DGB	B	Deglitch Selector B bit
DHALA	B	DAC Holdoff On All Listener Addresses bit
DHALL	B	DAC Holdoff On All bit
DHATA	B	DAC Holdoff On All Talker Addresses bit
DIR	R	Data In Register
DONE	B	Done bit
DRQ	B	Direct Memory Access Request Pin Status bit
DTAS	ST	Device Trigger Active State

E

Mnemonic	Type	Definition
EABO		Error Aborted
EAQ	Q	Error Active Queue
EARG		Error Argument
EIQ	Q	Error Input Queue
END	B	End Detected bit
ENOL		Error No Listeners
ENDONEOS	B	End On EOS bit
EOI	RM	End-or-Identify
EOI	B	End-or-Identify bit
EOIP		Error Operation In Progress
EOQ	Q	Error Output Queue
EOS	B	End-of-String Detected bit
EOSR	R	End-of-String Register
ERR	B	Error bit
ERR	RM	Error
ESB	B	Event Status bit
ESE		Event Status Byte (ESB) Enable
ESR	R	Event Status Register

Mnemonic	Type	Definition

F

Mnemonic	Type	Definition
FIFOs	R	FIFO A and FIFO B registers

G

Mnemonic	Type	Definition
GET	C	Group Execute Trigger multiline interface command
GLINT	B	Global Interrupt Enable bit
GO2SIDS	B	Go To SIDS bit
GTL	C	Go To Local command

H

Mnemonic	Type	Definition
HIER	R	High-Speed Enable Register
HLDA	B	Holdoff On All bit
HLDE	B	Holdoff On End bit
HSC		High-Speed Configure
HSE	B	High-Speed Enable bit
HSSEL	R	Handshake Select Register

I

Mnemonic	Type	Definition
IDAV	B	Input Data Bytes Available bit
IFC	B	Interface Clear bit
IMAV	B	Input Message/Event Available bit
IMBF	B	Input Message Buffer Full bit
IMR0	R	Interrupt Mask Register 0
IMR1	R	Interrupt Mask Register 1
IMR2	R	Interrupt Mask Register 2
IMR3	R	Interrupt Mask Register 3
IN	B	Input GPIB Data bit
INTR	R	Board Interrupt Register
INTSRC2	B	Interrupt Source 2 bit
IQAC	B	Input Queue Active bit
IQE	B	Input Queue Empty bit
IQEN	B	Input Queue Enabled bit
IQF	B	Input Queue Full bit
ISAV	B	Input Space Available bit
ISR0	R	Interrupt Status Register 0
ISR1	R	Interrupt Status Register 1
ISR2	R	Interrupt Status Register 2

Mnemonic	Type	Definition
ISR3	R	Interrupt Status Register 3
IST	B	Individual Status bit

K

Mnemonic	Type	Definition
KEYREG	R	Key Control Register

L

Mnemonic	Type	Definition
LA	B	Listener Active bit
LACS	B	Listener Active State bit
LLO	B	Local Lockout bit
LOK	B	Local Lockout bit
LON	B	Listen-Only bit
lon	LM	Listen Only

M

Mnemonic	Type	Definition
MAV	B	Message Available bit
MISC	R	Miscellaneous Register
MLA	RM	My Listen Address
MSA	RM	My Secondary Address
MTA	RM	My Talk Address

N

Mnemonic	Type	Definition
NACS	B	Not Active State bit
NDAC	B	Not Data Accepted bit
NEF	B	Not Empty FIFO bit
NFF	B	Not Full FIFO bit
NO_TSETUP	B	No TSETUP Delay bit
NRFD	RM	Not Ready For Data Message
NTNL	B	No Talking When No Listener bit

Mnemonic	Type	Definition
O		
ODWA	B	Output Data Bytes Waiting bit
OMBF	B	Output Message Buffer Full bit
OMWA	B	Output Message Waiting bit
OQAC	B	Output Queue Active bit
OQE	B	Output Queue Empty bit
OQEN	B	Output Queue Enabled bit
OQF	B	Output Queue Full bit
OSAV	B	Output Space Available bit
P		
PPR	R	Parallel Poll Register
R		
reg	R	Register
REM	B	Remote Programming bit
RESET FIFO	B	Reset FIFO bit
rhdf	B	Release Holdoff bit
rhdf	C	Release Holdoff
RL	F	Remote/Local
RQS	B	Requesting Service bit
S		
SDC	RM	Selected Device Clear
SH_CNT	R	SH_CNT Register
SIDS	ST	Source Idle State
SISB	B	Static Interrupt Status bit
SPMR	R	Serial Poll Mode Register
SPOLL	B	Serial Poll Active bit
SRE		Service Request Enable
SRQ	RM	Service Request
STB		Serial Poll Byte
STB	R	Status Byte Register
STBO	B	Status Byte Out bit

Mnemonic	Type	Definition
T		
T11	R	T11 Register
T12	R	T12 Register
T17	R	T17 Register
TA	B	Talker Active bit
TACS	B	Talker Active State bit
TIMER	R	Timer Register
TIMO	B	Time Limit Exceeded bit
TLCHTE	B	Halt On A Talker/Listener Interrupt bit
TLCINT	B	Talker/Listener Interrupt bit
TMOE	B	Timer Timeout Enable bit
TO	B	Timeout bit
ton	LM	Talk Only
TON	B	Talk-Only bit
TRI	B	Three-State Timing bit
U		
UNL	IM	Unlisten multiline interface message
UNT	IM	Untalk multiline interface message
USTD	B	Ultra Short T1 Delay bit
X		
XEOIWEOS	B	Transmit EOI With EOS bit

Appendix D
Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the *Technical Support Form* before contacting National Instruments helps us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, applications engineers are available Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

Corporate Headquarters
(512) 795-8248
Technical support fax: (512) 794-5678

Branch Offices	**Phone Number**	**Fax Number**
Australia	03 9 879 9422	03 9 879 9179
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	519 622 9310	
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	90 527 2321	90 502 2930
France	1 48 14 24 24	1 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	95 800 010 0793	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	01635 523545	01635 523154

Technical Support Form

Technical support is available at any time by fax. Include the information from your configuration form. Use additional pages if necessary.

Name ____

Company ____

Address ____

____

Fax (____)________________ Phone (____)________________________

Computer brand ____

Model __________________ Processor ______________________

Operating system ____

Speed ____________________MHz RAM ______________________MB

Display adapter ____

Mouse ___________yes __________no

Other adapters installed_

Hard disk capacity ___________MB Brand ______________________

Instruments used ____

National Instruments hardware product model ______________________

Revision ____

Configuration ____

(continues)

National Instruments software product ______________________________

Version ______________________________

Configuration ______________________________

The problem is ______________________________

____

____

____

____

____

____

____

List any error messages ______________________________

____

____

____

____

____

____

____

The following steps will reproduce the problem ______________________________

____

____

____

____

____

____

ESP-488TL Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Update this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration.

National Instruments Products

- ESP Model and Revision ______________________________
- NI-488.2M Software Revision Number on Disk______________________
- Parallel Port Configuration:______________________________
- Standard (factory setting) ______________________________
- PC ______________________________

Other Products

- Computer Make and Model ______________________________
- Memory Capacity on Computer ______________________________
- Operating System Version ______________________________
- Number of GPIB Devices on Bus ______________________________
- Other Hardware Devices in System ______________________________
- Type of Monitor ______________________________

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: **ESP-488TL Software Reference Manual for TNT4882™**

Edition Date: **July 1996**

Part Number: **370874A-01**

Please comment on the completeness, clarity, and organization of the manual.

(continues)

If you find errors in the manual, please record the page numbers and describe the errors.

____

____

____

____

____

____

____

____

____

____

____

Thank you for your help.

Name ____

Title ____

Company ____

Address ____

____

Phone (_____) _______________

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
(512) 794-5678

Glossary

Prefix	Meaning	Value
c-	cent-	10^{2}
G-	giga-	10^{9}
K-	kilo-	10^{3}
μ-	micro-	10^{-6}
m-	milli-	10^{-3}
M-	mega-	10^{6}
n-	nano-	10^{-9}

%	percent
ANSI	American National Standards Institute
ASIC	application-specific integrated circuit
CPU	central processing unit
DMA	direct memory access
EISA	Extended Industry Standard Architecture
EMI	electromagnetic interference
ESP	Engineering Software Package
GB	gigabytes
GPIB	General Purpose Interface Bus
hex	hexadecimal
Hz	hertz
IEEE	Institute of Electrical and Electronic Engineers
in.	inches
I/O	input/output
ISA	Industry Standard Architecture
KB	kilobytes
m	meters
MB	megabytes of memory
Mbytes	1,000,000 bytes
PC	personal computer
s	seconds
TL	Talker/Listener

Index

Numbers

16BIT bit, 6-12, 6-13

A

Abort_ASYNC_IO() function
definition, 2-10
description, 3-2
Address Mode Register (ADMR), 6-2
Address Register (ADR), 6-2
address variables. *See* CURRENT_ADDRESS variable; MULTI_ADDRESS variable.
addressing examples
normal I/O and multiple addressing
interrupt ESP, B-3 to B-4
noninterrupt ESP, B-11 to B-12
normal I/O and normal addressing
interrupt ESP, B-2 to B-3
noninterrupt ESP, B-9 to B-10
queues usage and multiple addressing (interrupt ESP), B-7 to B-8
simple queue usage and normal addressing (interrupt ESP), B-5 to B-6
addressing functions, GPIB
Change_Multiple_Addresses(), 2-10, 3-4 to 3-5
Change_Primary_Address(), 2-10, 3-3
Change_Secondary_Address(), 2-10, 3-6
Set_Address_Mode(), 2-10, 3-23 to 3-24
addressing modes
listen-only/talk-only addressing, 6-8 to 6-9
multiple primary addressing, 6-9 to 6-10
multiple primary and multiple secondary addressing, 6-10 to 6-11
single primary addressing, 6-6 to 6-7
single primary-multiple secondary addressing, 6-7 to 6-8
single primary-single secondary addressing, 6-7
addressing parameters for interrupt and noninterrupt ESP (table), A-2
ADDRESS_MODE parameter (table), A-2
ASYNC bit, 2-3
Auxiliary Mode Register
chip reset, 6-1
writing AUXMR setups, 6-2
Auxiliary Register B (AUXRB), 6-3

C

C language library. *See* GPIB C language library.
CABLE_LENGTH parameter (table), A-3
CCEN bit, 6-13
CF command, detecting, 6-3 to 6-4
Change_Multiple_Addresses() function
definition, 2-10
description, 3-4 to 3-5
Change_Primary_Address() function
definition, 3-3
description, 2-10
Change_Secondary_Address() function
definition, 2-10
description, 3-6

cleaning up I/O: DONE_Handler() & DONE_Interrupt_Handler(), 6-20 to 6-21
Clear_4882_Status() function
definition, 2-9
description, 3-7
Clear_Interrupt_Function() function
definition, 2-11
description, 3-8
coding conventions, ESP, 6-32 to 6-33
Command Pass Through Register (CPTR), 6-3
compiler options for C device programs
interrupt-driven ESP package, 2-8
noninterrupt-driven ESP package, 2-7
Configuration Register (CFG), 6-12
configuring interrupt and noninterrupt ESP. *See* interrupt and noninterrupt ESP.
Count Registers, 6-12
count variable. *See* DATA_COUNT variable.
CPT ENAB bit, 6-3
CPT Interrupt Status bit, 6-3
CURRENT_ADDRESS variable
definition, 2-1
description, 2-6
customer communication, *xiii*, D-1

D

Data Valid (DAV) deglitching value, 6-1, 6-4
Data_Available() function
definition, 4-10
description, 5-2
DATA_COUNT variable
definition, 2-1
description, 2-6
DCAS bit
clearing, 6-28
purpose and use, 2-4
DEC_Interrupt_Handler() function
definition, 4-4
description, 6-28
design considerations for programming (table), 1-3
DET_Interrupt_Handler() function
definition, 4-4
description, 6-28
Device Clear (DEC). *See* DEC_Interrupt_Handler() function.
Device Trigger (DET). *See* DET_Interrupt_Handler() function.
Disable_Input_Queue() function
definition, 4-11
description, 5-3
Disable_Output_Queue() function
definition, 4-11
description, 5-4
DMA I/O. *See* interrupt and noninterrupt I/O.
DMA_CHANNEL parameter (table), A-6
documentation
conventions used in manual, *xii*
organization of manual, *xi-xii*
related documentation, *xiii*
DONE bit, 6-14 to 6-17, 6-19, 6-21, 6-23
DONE_Interrupt_Handler() and DONE_Handler()
cleaning up I/O, 4-9, 6-20 to 6-21
optimizing I/O, 6-21 to 6-24
DTAS bit
clearing, 6-28

purpose and use, 2-4

E

EABO code, 2-5
EAQ code, 2-6
EARG code, 2-5
EIQ code, 2-5
Enable_GPIB_Interrupts() function, 6-24
Enable_Input_Queue() function
  definition, 4-11
  description, 5-5
Enable_Output_Queue() function
  definition, 4-11
  description, 5-6
END bit, 2-3
END message, 2-6
ENOL code, 2-5
EOIP code, 2-5
EOQ code, 2-6
EOS bit, 2-3
EOS character, 2-6
ERR bit
  clearing, 6-27
  purpose and use, 2-2
  error variable. *See* INTERFACE_ERROR variable.
ESE (Event Status Enable) Register, 6-31
ESP functions and utilities. *See* GPIB function and utility descriptions; queue function and utility descriptions.
ESP software. *See* also programming.
  code considerations, 1-3
  compiling C device program
    interrupt ESP package, 2-8
    noninterrupt ESP package, 2-7
  design considerations (table), 1-3
  files on distribution diskette, 1-1
  hardware considerations, 1-2
  kit contents, 1-1
  programming considerations, 1-2 to 1-3
ESR (Event Status Register), 6-31
event handling
  clearing INTERFACE_STATUS bits, 6-27 to 6-28
  creating new interrupt event handlers, 6-28 to 6-29
  including events in input queue, 6-28 to 6-29
  incomplete functions for handling DCAS, DTAS, SPOLL, and IFC (note), 2-4
  input queue and events, 4-9
Event Status Enable (ESE) Register, 6-31
Event Status Register (ESR), 6-31

G

global variables
  CURRENT_ADDRESS, 2-6
  DATA_COUNT, 2-6
  INTERFACE_ERROR, 2-5 to 2-6
  INTERFACE_STATUS, 2-2 to 2-4
  list of variables, 2-1
  MULTI_ADDRESS, 2-6
  QUEUE_ADDRESS, 4-7
  QUEUE_COUNT, 4-7
  QUEUE_STATUS, 4-5 to 4-7
GO bit, 6-12
GPIB C language library
  compiling C device program
    interrupt ESP package, 2-8
    noninterrupt ESP package, 2-7
  function descriptions. *See* GPIB function and utility descriptions.

global variables
CURRENT_ADDRESS, 2-6
DATA_COUNT, 2-6
INTERFACE_ERROR, 2-5 to 2-6
INTERFACE_STATUS, 2-2 to 2-4
list of variables, 2-1
MULTI_ADDRESS, 2-6
GPIB function and utility descriptions
addressing functions
Change_Multiple_Addresses(), 2-10, 3-4 to 3-5
Change_Primary_Address(), 2-10, 3-3
Change_Secondary_Address(), 2-10, 3-6
Set_Address_Mode(), 2-10, 3-23 to 3-24
I/O functions
Abort_ASYNC_IO(), 2-10, 3-2
Read_GPIB_Lines(), 2-10, 3-13
Receive(), 2-10, 3-14 to 3-15
Receive_ASYNC(), 2-10, 3-16 to 3-17
Send(), 2-10, 3-18 to 3-19
Send_ASYNC(), 2-10, 3-20 to 3-21
initialization functions
High_Speed_Select(), 2-9, 3-9
Initialize_Interface(), 2-9, 3-10
Interface_Off(), 2-9, 3-11
Set_Timeout(), 2-9, 3-27 to 3-28
interrupt control functions
Clear_Interrupt_Function(), 2-11, 3-8
Set_Interrupt_Function(), 2-11, 3-25 to 3-26
status functions
Clear_4882_Status(), 2-9, 3-7
Read_4882_Status(), 2-9, 3-12
Set_4882_Status(), 2-9, 3-22
Update_INTERFACE_STATUS(), 2-9, 3-29 to 3-30
Wait_For_Interface(), 2-9, 3-31
GPIB I/O with TNT4882
cleaning up I/O: DONE_Handler() & DONE_Interrupt_Handler(), 6-20 to 6-21
interrupt and noninterrrupt DMA I/O: GPIB_DMA_IO(), 6-17 to 6-20
interrupt and noninterrupt programmed I/O: GPIB_PROG_IO(), 6-14 to 6-16
handling noninterrupt programmed I/O, 6-14 to 6-16
important status bits, 6-14
interrupt-driven programmed I/O, 6-16 to 6-17
optimizing I/O, 6-21 to 6-24
avoiding interrupts, 6-23
checking assembled code, 6-24
compiling fast code, 6-24
DONE_Interrupt_Handler() and DONE_Handler(), 6-22 to 6-23
eliminating loops, 6-22
eliminating unneccessary tasks, 6-22
I/O testing, 6-22
Interrupt_Handler(), 6-23
programming examples
normal I/O and multiple addressing
interrupt ESP, B-3 to B-4
noninterrupt ESP, B-11

to B-12

normal I/O and normal addressing
interrupt ESP, B-2 to B-3
noninterrupt ESP, B-9 to B-10
setting up for I/O: Setup_TNT_IO(), 6-11 to 6-13
input, 6-12
output, 6-13
GPIB_DMA_IO() source code, 6-17 to 6-20
GPIB_PROG_IO() source code, 6-14 to 6-16

H

handlers, miscellaneous, for interrupt and noninterrupt ESP (table), A-4 to A-5
handshake mode and high-speed I/O, configuring, 6-3 to 6-6
detecting CF command, 6-3 to 6-4
Handshake Select Register (HSSEL), 6-1, 6-5
Mode and Timing register configuration, 6-4 to 6-6
normal three-wire handshake mode, 6-5
setting TNT4882 to HS488, 6-5 to 6-6
Timing Registers configuration table, 6-4
hardware parameters for interrupt and noninterrupt ESP (table), A-6
header files
interrupt-driven ESP package, 2-7
noninterrupt-driven ESP package, 2-7
High-Speed Enable Register (HIER), 6-5
high-speed I/O mode. *See* handshake mode and high-speed I/O, configuring.
High_Speed_Select() function, 2-9, 3-9
HS488 capabilities of TNT4882
enabling, 6-3
setting, 6-5 to 6-6
HS_MODE parameter (table), A-3

I

I/O functions
GPIB
Abort_ASYNC_IO(), 2-10, 3-2
Read_GPIB_Lines(), 2-10, 3-13
Receive(), 2-10, 3-14 to 3-15
Receive_ASYNC(), 2-10, 3-16 to 3-17
Send(), 2-10, 3-18 to 3-19
Send_ASYNC(), 2-10, 3-20 to 3-21
queue
Disable_Input_Queue(), 4-11, 5-3
Disable_Output_Queue(), 4-11, 5-4
Enable_Input_Queue(), 4-11, 5-5
Enable_Output_Queue(), 4-11, 5-6
Read_Input_Queue(), 4-11, 5-13 to 5-14
Write_Output_Queue(), 4-10, 5-17
I/O parameters for interrupt and noninterrupt ESP (table), A-3
I/O programming examples

normal I/O and multiple addressing
interrupt ESP, B-3 to B-4
noninterrupt ESP, B-11 to B-12
normal I/O and normal addressing
interrupt ESP, B-2 to B-3
noninterrupt ESP, B-9 to B-10
I/O queues. *See* queue function and utility descriptions; queues.
I/O with TNT4882. *See* GPIB I/O with TNT4882.
IDAV bit, 4-6, 4-9
IEEE 488.2 status model
adding new registers, 6-31 to 6-32
required IEEE 488.2 Status registers, 6-31
IFC bit
clearing, 6-27
purpose and use, 2-3
IMAV bit, 4-5, 4-9
IMBF bit, 4-6
IMR0, IMR1, and IMR2 interrupts, enabling, 6-25
IN bit, 6-12
INCLUDE_EVENT_HANDLERS constant, 4-4, 4-9, 6-28
initialization functions
GPIB
High_Speed_Select(), 2-9, 3-9
Initialize_Interface(), 2-9, 3-10
Interface_Off(), 2-9, 3-11
Set_Timeout(), 2-9, 3-27 to 3-28
queue
Initialize_Input_Queue(), 4-10, 5-7
Initialize_Output_Queue(), 4-10, 5-8
initialization of TNT4882
detecting CF command, 6-3 to 6-4
handshake mode and high speed I/O, 6-3
Mode and Timing registers, 6-4 to 6-5
setting to HS488, 6-5 to 6-6
setting to normal three-wire handshake mode, 6-5
steps, 6-1 to 6-3
Timing registers configuration table, 6-4
Initialize_Input_Queue() function
definition, 4-10
description, 5-7
Initialize_Interface() function, 6-28
definition, 2-9
description, 3-10
effect of USE_QUEUES parameter, 4-8
Initialize_Output_Queue() function
definition, 4-10
description, 5-8
input queue. *See* queues.
INPUT_QUEUE_SIZE parameter
definition (table), 4-8, A-7
setting queue size, 4-3
INTSRC2 bit, 6-14 to 6-16
INTERFACE_ERROR variable
definition, 2-1
EABO code, 2-5
EAQ code, 2-6
EARG code, 2-5
EIQ code, 2-5
ENOL code, 2-5
EOIP code, 2-5
EOQ code, 2-6
error code descriptions (table), 2-5
Interface_Off() function

definition, 2-9
description, 3-11
INTERFACE_STATUS variable, 2-2 to 2-4
ASYNC bit, 2-3
clearing, 6-27 to 6-28
DCAS bit, 2-4, 6-28
definition, 2-1
DTAS bit, 2-4, 6-28
END bit, 2-3
EOS bit, 2-3
ERR bit, 2-2, 6-27
IFC bit, 2-3, 6-27
incomplete functions for handling DCAS, DTAS, SPOLL, and IFC (note), 2-4
LACS bit, 2-4
list of bits (table), 2-2
LOK bit, 2-3
NACS bit, 2-4
REM bit, 2-3
RQS bit, 2-3
SPOLL bit, 2-3, 6-27
TACS bit, 2-4
TIMO bit, 2-3
UCMPL bit, 2-3
interrupt and noninterrupt ESP
compiling C device program
interrupt ESP package, 2-8
noninterrupt ESP package, 2-7
configuring, A-1 to A-7
addressing parameters (table), A-2
hardware parameters (table), A-6
I/O parameters (table), A-3
miscellaneous handlers (table), A-4 to A-5
queue model parameters (table), A-7
software configuration (table), A-7
interrupt and noninterrupt I/O
cleaning up I/O, 6-20 to 6-21
DMA I/O: GPIB_DMA_IO(), 6-17 to 6-20
interrupt-driven DMA I/O, 6-19 to 6-20
noninterrupt DMA I/O with Intel 8237 DMA Controller, 6-18 to 6-19
interrupt-driven programmed I/O, 6-16 to 6-17
programmed I/O: GPIB_PROG_IO(), 6-14 to 6-16
handling noninterrupt programmed I/O, 6-14 to 6-16
important status bits, 6-14
interrupt control functions, GPIB
Clear_Interrupt_Function(), 2-11, 3-8
Set_Interrupt_Function(), 2-11, 3-25 to 3-26
Interrupt Status Register 1 (ISR1), 6-3
INTERRUPT_CHANNEL parameter (table), A-6
Interrupt_Handler(). *See also* interrupts.
calling of Queue_Interrupt_Handler(), 4-8
enabling, 6-24 to 6-25
handling of interrupt routine, 6-26 to 6-27
optimizing, 6-23
interrupts, 6-24 to 6-27
avoiding, 6-23

calling of interrupt routine by Interrupt_Handler(), 6-26 to 6-27
clearing interrupt condition, 6-26
detection of interrupt condition, 6-25 to 6-27
enabling, 6-24 to 6-25
IMR0, IMR1, and IMR2, 6-25
interrupt handler, 6-24 to 6-25
ISR3, 6-25
incomplete functions for handling DCAS, DTAS, SPOLL, and IFC (note), 2-4
io_param.h queue configuration constants (table), 4-8
IQAC bit, 4-6
IQE bit, 4-5
IQEN bit, 4-6
IQF bit, 4-5
ISAV bit, 4-6
ISR3 interrupt, enabling, 6-25

L

LACS bit, 2-4
listen-only/talk-only addressing, 6-8 to 6-9
LOK bit, 2-3

M

manual. *See* documentation.
MAXIMUM_BUFFER_SIZE parameter
definition (table), 4-8
setting number of terminated messages, 4-3
message entry, 4-3, 4-4
Message_Available() function
definition, 4-10
description, 5-9
MESSAGE_BUFFER_SIZE parameter (table), A-7
Message_Length() function
definition, 4-10
description, 5-10
mnemonics key, C-1 to C-7
Mode and Timing Registers, 6-4 to 6-5
MULTI_ADDRESS variable
definition, 2-1
description, 2-6
multichip communications, 6-29 to 6-30
multiple primary addressing, 6-9 to 6-10
multiple primary and multiple secondary addressing, 6-10 to 6-11
MULTIPLE_ADDRESSES parameter (table), A-2

N

NACS bit, 2-4
NEF bit, 6-14 to 6-16
NFF bit, 6-14 to 6-16
noninterrupt ESP package. *See* interrupt and noninterrupt ESP.
noninterrupt I/O. *See* interrupt and noninterrupt I/O.
NUMBER_OF_ADDRESSES parameter (table), A-2

O

ODWA bit, 4-6, 4-9
OMBF bit, 4-6 to 4-7
OMWA bit, 4-6, 4-9
optimizing I/O, 6-21 to 6-24
avoiding interrupts, 6-23
checking assembled code, 6-24
compiling fast code, 6-24
DONE_Interrupt_Handler() and

DONE_Handler(), 6-22 to 6-23
eliminating loops, 6-22
eliminating unneccessary tasks, 6-22
I/O testing, 6-22
Interrupt_Handler(), 6-23
OQAC bit, 4-7
OQE bit, 4-6
OQEN bit, 4-7
OQF bit, 4-6
OSAV bit, 4-7
output queue. *See* queues.
OUTPUT_QUEUE_SIZE parameter
definition (table), 4-8, A-7
setting queue size, 4-3
Output_Space_Available() function
definition, 4-10
description, 5-11 to 5-12

P

Parallel Poll Register (PPR), 6-2
primary address. *See* addressing functions, GPIB; addressing modes.
PRIMARY_ADDRESS parameter (table), A-2
programming
addressing modes
listen-only/talk-only addressing, 6-8 to 6-9
multiple primary addressing, 6-9 to 6-10
multiple primary and multiple secondary addressing, 6-10 to 6-11
single primary addressing, 6-6 to 6-7
single primary-multiple secondary addressing, 6-7 to 6-8
single primary-single secondary addressing, 6-7
compiling C device programs
interrupt ESP, 2-8
noninterrupt ESP, 2-7
design considerations (table), 1-3
ESP coding conventions, 6-32 to 6-33
ESP software considerations, 1-2 to 1-3
event handling
clearing
INTERFACE_STATUS bits, 6-27 to 6-28
creating new interrupt event handlers, 6-28 to 6-29
including events in input queue, 6-28 to 6-29
examples
basic device program model, B-1
normal I/O and multiple addressing
interrupt ESP, B-3 to B-4
noninterrupt ESP, B-11 to B-12
normal I/O and normal addressing
interrupt ESP, B-2 to B-3
noninterrupt ESP, B-9 to B-10
queues usage and multiple addressing (interrupt ESP), B-7 to B-8
simple queue usage and normal addressing (interrupt ESP), B-5 to B-6
GPIB I/O with TNT4882
cleaning up I/O, 6-20 to 6-21

interrupt and noninterrrupt DMA I/O, 6-17 to 6-20

interrupt and noninterrupt programmed I/O, 6-14 to 6-16
interrupt-driven programmed I/O, 6-16 to 6-17
optimizing I/O, 6-21 to 6-24
setting up for I/O, 6-11 to 6-13
IEEE 488.2 status model
adding new registers, 6-31 to 6-32
required IEEE 488.2 Status registers, 6-31
interrupts and ESP software
detection of interrupt condition, 6-25 to 6-27
enabling interrupts, 6-24 to 6-25
multichip communications, 6-29 to 6-30
TNT4882 initialization and configuration
detecting CF command, 6-3 to 6-4
handshake mode and high speed I/O, 6-3
Mode and Timing registers, 6-4 to 6-5
setting to HS488, 6-5 to 6-6
setting to normal three-wire handshake mode, 6-5
steps, 6-1 to 6-3
Timing registers configuration table, 6-4

Q

queue function and utility descriptions
I/O functions
Disable_Input_Queue(), 4-11, 5-3
Disable_Output_Queue(), 4-11, 5-4
Enable_Input_Queue(), 4-11, 5-5
Enable_Output_Queue(), 4-11, 5-6
Read_Input_Queue(), 4-11, 5-13 to 5-14
Write_Output_Queue(), 4-10, 5-17
initialization functions
Initialize_Input_Queue(), 4-10, 5-7
Initialize_Output_Queue(), 4-10, 5-8
status functions
Data_Available(), 4-10, 5-2
Message_Available(), 4-10, 5-9
Message_Length(), 4-10, 5-10
Output_Space_Available(), 4-10, 5-11 to 5-12
Wait_For_Queue(), 4-10, 5-15 to 5-16
queue model parameters for interrupt ESP (table), A-7
QUEUE_ADDRESS global variable
input queue handling, 4-9
purpose and use, 4-7
QUEUE_COUNT global variable
input queue handling, 4-9
purpose and use, 4-7
Queue_Interrupt_Handler() function
accessing data in queues, 4-3
input queue handling, 4-9
output queue handling, 4-9

queue operation, 4-8
queues
configuration, 4-8
information globals
QUEUE_ADDRESS, 4-7
QUEUE_COUNT, 4-7
QUEUE_STATUS, 4-5 to 4-7
input queue handling, 4-9
input queues and events, 4-9
io_param.h file (table), 4-8
models, 4-2 to 4-3
illustration, 4-2
operation of, 4-8 to 4-9
output queue handling, 4-9
programming examples
queues usage and multiple addressing (interrupt ESP), B-7 to B-8
simple queue usage and normal addressing (interrupt ESP), B-5 to B-6
structure, 4-3 to 4-4
illustration, 4-3
QUEUE_STATUS status word
bits and bit positions (table), 4-5, 5-15
IDAV bit, 4-6, 4-9
IMAV bit, 4-5, 4-9
IMBF bit, 4-6
IQAC bit, 4-6
IQE bit, 4-5
IQEN bit, 4-6
IQF bit, 4-5
ISAV bit, 4-6
ODWA bit, 4-6, 4-9
OMBF bit, 4-6 to 4-7
OMWA bit, 4-6, 4-9
OQAC bit, 4-7
OQE bit, 4-6
OQEN bit, 4-7
OQF bit, 4-6
OSAV bit, 4-7

R

read and write termination, 2-6 to 2-7
END message, 2-6
EOS character, 2-6
READ_EOS_BYTE constant, 2-7
WRITE_EOS_BYTE constant, 2-7
Read_4882_Status() function
definition, 2-9
description, 3-12
READ_EOS_BYTE constant, 2-7
READ_EOS_BYTE parameter (table), A-3
Read_GPIB_Lines() function
definition, 2-10
description, 3-13
Read_Input_Queue() function
accessing data in queues, 4-3
definition, 4-11
description, 5-13 to 5-14
input queue handling, 4-11
Receive() function
definition, 2-10
description, 3-14 to 3-15
Receive_ASYNC() function
definition, 2-10
description, 3-16 to 3-17
REM bit, 2-3
RQS bit, 2-3

S

secondary address. *See* addressing functions, GPIB; addressing modes.

SECONDARY_ADDRESS parameter (table), A-2

Send() function
- definition, 2-10
- description, 3-18 to 3-19

Send_ASYNC() function
- definition, 2-10
- description, 3-20 to 3-21

Serial Poll Mode Register (SPMR), 6-1
Service Request Enable (SRE) Register, 6-31
Set_4882_Status() function
- definition, 2-9
- description, 3-22

Set_Address_Mode() function
- definition, 2-10
- description, 3-23 to 3-24

Set_Interrupt_Function() function
- definition, 2-11
- description, 3-25 to 3-26
- handling of interrupt routine, 6-26 to 6-27

Set_Timeout() function
- definition, 2-9
- description, 3-27 to 3-28

Setup_TNT_IO() function, 6-11 to 6-13
- input, 6-12
- output, 6-13

SH_CNT Register, 6-5
single primary addressing, 6-6 to 6-7
single primary-multiple secondary addressing, 6-7 to 6-8
single primary-single secondary addressing, 6-7
software configuration parameters, for interrupt and noninterrupt ESP (table), A-7
SPOLL bit
- clearing, 6-27
- purpose and use, 2-3

SRE (Service Request Enable) Register, 6-31
Status Byte Register (STB), 6-31
status functions
- GPIB
  - Clear_4882_Status(), 2-9, 3-7
  - Read_4882_Status(), 2-9, 3-12
  - Set_4882_Status(), 2-9, 3-22
  - Update_INTERFACE_STATUS(), 2-9, 3-29 to 3-30
  - Wait_For_Interface(), 2-9, 3-31
- queue
  - Data_Available(), 4-10, 5-2
  - Message_Available(), 4-10, 5-9
  - Message_Length(), 4-10, 5-10
  - Output_Space_Available(), 4-10, 5-11 to 5-12
  - Wait_For_Queue(), 4-10, 5-15 to 5-16

status registers (IEEE 488.2), 6-31 to 6-32
status variable. *See* INTERFACE_STATUS variable.
STB (Status Byte Register), 6-31
structure of queues, 4-3 to 4-4

T

TACS bit, 2-4
talk-only addressing, 6-8 to 6-9
technical support, D-1
TIMBYTN bit, 6-12, 6-13
TIMEOUT_FACTOR_INDEX parameter (table), A-3
Timing Register
- configuration table, 6-4

configuring Mode and Timing Registers, 6-4 to 6-5
TIMO bit, 2-3
TLCHTE bit, 6-12, 6-13
TLCINT bit, 6-14
TMOE bit, 6-12, 6-13
TNT4882 programming. *See* programming.
TNT4882 TL ESP software. *See* ESP software.
TNT_BASE_ADDRESS parameter (table), A-6

U

UCMPL bit, 2-3
Update_INTERFACE_STATUS() function
definition, 2-9
description, 3-29 to 3-30
Update_Queue_Info() function, 6-29
USE_BYTE_TIMEOUTS parameter (table), A-3
USE_DEMAND_MODE_DMA parameter (table), A-6
USE_DMA parameter
definition (table), A-6
input queue handling, 4-9
USE_EIGHT_BIT_EOS_COMPARE parameter (table), A-3
USE_EOS_WITH_QUEUES parameter (table), 4-8, A-7
USE_EVENT_HANDLERS handler (table), A-4
USE_HIGH_SPEED_T1 parameter (table), A-3
USE_HS488 handler (table), A-5
USE_HUGE_BUFFERS parameter (table), A-7
USE_MICROSOFT_C parameter (table), A-7
USE_QUEUES parameter (table), 4-8, A-7
User_GPIB_IO() function, 6-11
USE_SPOLL_BIT handler (table), A-4
USE_TRANSMIT_EOI_WITH_EOS parameter (table), A-3
utilities. *See* GPIB function and utility descriptions; queue function and utility descriptions.

W

Wait_For_Interface() function
definition, 2-9
description, 3-31
Wait_For_Queue() function
definition, 4-10
description, 5-15 to 5-16
write termination. *See* read and write termination.
WRITE_EOS_BYTE parameter
definition (table), A-3
purpose and use, 2-7
Write_Output_Queue() function
accessing data in queues, 4-3
definition, 4-10
description, 5-17
output queue handling, 4-9