

ines-ieee488.2

iGPIB Device Driver
Software Manual

© 1998 ines GmbH

All rights reserved. No part of this document may be reproduced without permission in writing from the publisher.

Hannover (Germany), October, 1998

Contents

1. Introduction	3
2. Initialization	3
3. Input Queue	3
3.1 Check Ready	3
3.2 Reading Data	4
4. Output Queue.....	4
4.1 Checking for Free Space.....	4
4.2 Writing Data.....	4
5. Device Status	4
6. Terminators	5
7. Status- and Service Request Enable Byte.....	5
8. Additional Routines	5
8.1 Power On	5
8.2 Release Holdoff	5
8.3 Timeout	6
8.4 Interrupt Handler	6
9. Application Services	6
9.1 iGPIB Register Access.....	7
9.2 Disabling Interrupts	7
9.3 Signal Handler	7
10. IO Buffer Structures	8

1. Introduction

This software module provides routines to the iGPIB controller in IEEE488.2 devices. It may also be used for systems which do not use the IEEE488.2 protocol, but require a GPIB stream-like interface.

The software module is written in ANSI compatible C and may be compiled for any micro processor.

Together with the iGPIB controller, a free-of-charge licence is provided.

The interface routines are declared in *igpib.h*. The routines that may be called by the application are declared as well as those routines which must be provided by the application.

The device software uses the input and output queue of the iGPIB which is 255 byte in depth each. This software does not supply a buffer mechanism to support amounts of data higher than 255 byte.

To get a maximum of portability, it is necessary to provide the method of how to access the iGPIB by the processor.

The file *igctrl.h* will allow to set up the following methods and values:

- Access to the iGPIB register, (IO-port or memory mapped)
- manufacturer, model number and serial number necessary for automatic addressing.

2. Initialization

To initialize the iGPIB, the following information is needed:

- physical address of the iGPIB,
- the primary and, perhaps, the secondary address and
- a boolean value if hardware trigger should be enabled.

```
int igInit(struct ig_init *myIgInit);
struct ig_init {
    PORT    physAddr;    /* physical port base address */
    int     primAddr;    /* ieee488 logical addr or -1 */
    int     secAddr;    /* secondary addr or -1 */
    int     hwTrigger;  /* enable hardware trigger */
};
```

The routine returns a non-zero value if the iGPIB was not found at the address specified (return value: IGERR_NFND) or if a not implemented feature¹ has been enabled (return value: IGERR_NOIMPL).

3. Input Queue

3.1 Check Ready

The routine

```
int igIQRdy(unsigned int cnt, int hSig);
```

checks if *cnt* bytes are in the input queue or if an End message has been received. In these cases, the actual number of bytes in the input queue will be returned. If a trigger has not been acknowledged yet, this routine will count this command as one byte and return its count. The input queue may be read through *igIQRead()*.

¹Automatic addressing is not yet implemented.

A zero will be returned if not enough bytes are available. In this case the variable *hSig* allows the following choices:

- if *hSig* == 0, no asynchronous routine will be called.
- if *hSig* != 0, the interrupts will be disabled through *igDisInt()* and the routine *igSignal(sigId, hSig)* will be called when *cnt* bytes are available or an END condition is detected.
The variable *sigId* will contain IGSIG_IQ and *hSig* will contain the value supplied on the *igIQRdy()* call.

3.2 Reading Data

The routine

```
int igIQRead(unsigned int cnt, struct ig_usrBuf *data);
```

reads data from the input queue and writes it into the user supplied buffer *data*. The structure *ig_usrBuf* is described below.

4. Output Queue

4.1 Checking for Free Space

The routine

```
int igOQRdy(unsigned int cnt, int hSig);
```

checks if *cnt* bytes may be written into the output queue. In this case the routine returns the amount of free space in the output queue.

A zero will be returned if not enough free space is available. In this case the variable *hSig* may be used to switch between the following choices:

- if *hSig* == 0, no asynchronous routine will be called.
- if *hSig* != 0, the interrupts will be disabled through *igDisInt()* and the routine *igSignal(sigId, hSig)* will be called when *cnt* bytes may be written into the output queue.
The variable *sigId* will contain IGSIG_OQ and *hSig* will contain the value supplied on the *igOQRdy()* call.

4.2 Writing Data

The routine

```
int igOQWrite(unsigned int cnt, struct ig_usrBuf *data);
```

sends *cnt* bytes from the user defined buffer *data* into the output queue. The struct *ig_usrBuf* is described below. The routine will return immediately when the bytes are put into the output queue. This does not mean that the bytes are transferred via the GPIB bus!

5. Device Status

To obtain the actual device interface status you may call the routine

```
int igDevStatus();
```

which returns an integer value. The following bits contain the status of the interface:

IGS_SRQS	The device is in service request state.
IGS_TADS	The device is in talker addressed state.
IGS_LADS	The device is in listener addressed state.
IGS_IFRDY	The input FIFO is ready. Data may be read now.
IGS_OFRDY	The output FIFO is ready. Data may be written now.
IGS_REMS	The device is in remote state.
IGS_LOKS	The device is in lockout state.

6. Terminators

Four routines are available to specify and read the actual terminators for in- and output. For input it is possible to specify up to 3 EOS characters and for output 1 EOS character is available.

```
void igSetTermIn(int eoi, int eos1, int eos2, int eos3);
void igGetTermIn(int *eoi, int *eos1, int *eos2, int *eos3);

void igSetTermOut(int eoi, int eos);
void igGetTermOut(int *eoi, int *eos);
```

7. Status- and Service Request Enable Byte

The status byte will be handled like specified in the IEEE488.2. A service request is generated if a transition of one bit from 0 to 1 is done in the status byte and the corresponding bit in the service request enable register is 1. The status byte may be read or written with the following routines:

```
void igSpStatWr(BYTE myStat);
BYTE igSpStatRd();
```

The service request enable register may be accessed analogously:

```
void igSpEnabWr(BYTE myMask);
BYTE igSpEnabRd();
```

According to IEEE488.2 the bit 4 (MAV: message available) is reset in the status byte if a message (a string terminated with EOI) has been sent completely. Bit 5 (ESB: event status byte) has to be managed by the software.

8. Additional Routines

8.1 Power On

To issue a Power On the routine

```
igPon();
```

may be called. The iGPIB will be reset to its init states but the configuration will not be lost.

8.2 Release Holdoff

The iGPIB software issues a DAC holdoff on reception of a clear command or a trigger command. If the *igSignal()* routine has been called with IGSIG_DACS or IGSIG_TRIG the GPIB bus will holdoff until

```
igAck();
```

is called by the application software.

8.3 Timeout

The routine

```
void igTmo(unsigned int rwTmo);
```

allows the application to set a timeout value on listener and talker transfers. A timeout will be marked by calling the routine *igSignal()* with IGSIG_RDTMO or IGSIG_WRTMO as parameter.

8.4 Interrupt Handler

The device software may be polled or may be interrupt driven. On occurrence of an interrupt, the application has to call the routine *igIntr()* to notify the device software. The return value is equal to 0 if the interrupt has not come from the iGPIB controller, otherwise it returns 1.

```
int igIntr();
```

If no hardware interrupt is used with the device software you may use a polling loop to check the occurrence of any changes. The routine

```
int igPoll();
```

has to be called to get the device software running. The routine will return 0 if no interrupt condition was found, otherwise it returns 1.

9. Application Services

The following routines must be provided by the application program. They are used to abstract the iGPIB device software from the existing hardware.

9.1 iGPIB Register Access

The iGPIB will be accessed by the routines *inb()* and *outb()* as declared below.

```
void outb(PORT ioAddr, BYTE oByte);
BYTE inb(PORT ioAddr);
```

The *ioAddr* will be generated from the physical address supplied with the initialization and the computed register offset. The type PORT has to be supplied in *igctrl.h* and reflects the method, how the iGPIB may be accessed by the processor (e.g. IO-port or memory mapped).

The routine *outb()* writes a byte at the given address and *inb()* reads a byte.

9.2 Disabling Interrupts

The routines *igIQRdy()* and *igOQRdy()* have to disable the interrupt so that they cannot be called recursively. This happens if the condition needed by the application is not met and the *hSig* parameter is not 0. Just before the interrupts are set the routine

```
void igDisInt();
```

is called. The application should atomically set up its own flags and enable the interrupts again.

9.3 Signal Handler

The signal handler is called to notify the application of an asynchronous event. The parameter *sigId* will hold an identification of the signal type and the parameter *hSig* depends on *sigId*.

```
void igSignal(int sigId, int hSig);
```

Signal	
IGSIG_IQ	The input queue has received the requested amount of data or a valid END message. The parameter <i>hSig</i> will contain the value supplied by the <i>igIQRdy()</i> call.
IGSIG_OQ	The output queue has enough room for the requested amount of data. The parameter <i>hSig</i> will contain the value supplied by the <i>igOQRdy()</i> call.
IGSIG_DCAS	A device clear has been received. The DAC holdoff must be released by the application. The output FIFO is cleared after returning from <i>igSignal()</i> .
IGSIG_TRIG	A device trigger has been received. The DAC holdoff must be released by the application.
IGSIG_LOC	The lockout state has changed. The parameter <i>hSig</i> is 0 if the interface is not in lockout state at now.
IGSIG_REM	The remote state has changed. The parameter <i>hSig</i> is 0 if the interface is in local state now, otherwise it is in remote state.
IGSIG_RDTMO	A timeout has occurred during a read operation.
IGSIG_WRTMO	A timeout has occurred during a write operation.
IGSIG_INTR	The IEEE488.2 defined INTERRUPTED condition has occurred. This means that the controller has addressed the interface as listener while a message has not been sent completely.
IGSIG_DEAD	The IEEE488.2 defined DEADLOCK condition has been met. In this case the output queue is full and the application software tries to respond to a full input queue while this queue has become full.

10. IO Buffer Structures

For data input/output the following structure is used:

```
struct ig_usrBuf() {  
    BYTE    *data;        /* data bytes to be transfered */  
    BYTE    *ctrl;       /* ctrl array or NULL */  
    BYTE    inopt;       /* read until END condition */  
    BYTE    outopt;      /* append END */  
};
```

data	The 8 bit wide device dependent messages are stored in this array. The buffer must be delivered by the application for both in- and output. The array must be as long as the <i>cnt</i> parameter specified by the IO operation call.
Ctrl	The <i>ctrl</i> pointer may be NULL if it is not used. Otherwise the END message, indicated by the value IQ_EOI, or the trigger message, indicated by the value IQ_TRG, will be found at the corresponding position. The <i>ctrl</i> array if it exists has to be as long as the <i>cnt</i> variable, specified by the IO operation.
Inopt	If <i>inopt</i> is 1 the <i>igIQRead()</i> will stop until an END message has been detected or a trigger has been found.
Outopt	if <i>outopt</i> is 1 the actual data will be sent with the actual terminator.