

# CDI

## Application Program Interface Guide

March 1999



© 1997-1999 GCOM, Inc. All rights reserved.

Non-proprietary—Provided that this notice of copyright is included, this document may be copied in its entirety without alteration. Permission to publish excerpts should be obtained from GCOM, Inc.

Rsystem is a registered trademark of GCOM, Inc. UNIX is a registered trademark of UNIX Systems Laboratories, Inc. in the U.S. and other countries. SCO is a trademark of the Santa Cruz Operation, Inc. All other brand product names mentioned herein are the trademarks or registered trademarks of their respective owners.

Any provision of this product and its manual to the U.S Government is with “Restricted Rights”: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013 of the DOD FAR Supplement.

This manual was written and produced by Technical Writer Geoffrey Gerrietts using Framemaker 5.01 and Microsoft Word 97 on a Microsoft Windows platform with the help of subject matter specialists Dave Grothe, Mikel Matthews, and Dave Healy.

This manual was printed in the U.S.A.

**FOR FURTHER INFORMATION**

If you want more information about GCOM products, contact us at:

GCOM, Inc.  
1800 Woodfield  
Savoy, IL 61874  
(217) 337-4471  
FAX: (217) 337-4470  
e-mail: [support@gcom.com](mailto:support@gcom.com)  
homepage: <http://gcom.com>



# C O N T E N T S

---

## SECTION 1

The CDI API 7  
Introduction 8  
Using the CDI API 9  
Preparing STREAMS 9  
Manipulating the Device 10  
Data Communications 12  
Utility Routines 14  
Global Variables 15  
Decoding Control Messages from the CDI Provider 15

## SECTION 2

API Reference 17  
cdi\_allow\_input\_req() 19  
cdi\_attach\_req() 20  
cdi\_decode\_ctl() 21  
cdi\_decode\_modem\_sigs() 22  
cdi\_detach\_req() 23  
cdi\_disable\_req() 24  
cdi\_enable\_req() 25  
cdi\_get\_a\_msg 26  
cdi\_get\_modem\_sigs 27  
cdi\_init() 28  
cdi\_init\_FILE() 29  
cdi\_modem\_sig\_poll 30  
cdi\_modem\_sig\_req() 31  
cdi\_open\_data() 32  
cdi\_perror() 33  
cdi\_printf() 34  
cdi\_print\_msg() 35  
cdi\_put\_allow\_input\_req() 36  
cdi\_put\_attach\_req() 37  
cdi\_put\_both() 38  
cdi\_put\_data() 39  
cdi\_put\_detach\_req() 40  
cdi\_put\_disable\_req() 41  
cdi\_put\_enable\_req() 42  
cdi\_put\_frame() 43  
cdi\_put\_proto() 44  
cdi\_rcv\_msg() 45  
cdi\_read\_data() 46  
cdi\_set\_log\_size() 47

cdi\_wait\_ack() 48  
cdi\_write\_data() 49  
cdi\_xray\_req() 50

**APPENDIX A**

Sample Decoder 67

# 1

---

## *The CDI API*

This section of the manual identifies the components of the CDI API and describes how an application can use the API to create communications applications.

## *Introduction*

The original AT&T specification for the STREAMS facility included three well-defined interfaces: TLI, the transport layer interface; NPI, the network provider interface; and DLPI, the data-link provider interface. These three interfaces covered the OSI model's layers 4 down to 2. DLPI was assumed to be the lowest level software interface, leaving the 'physical' layer functions completely in the hands of the hardware.

While this works well with many popular networking strategies, it doesn't work with all of them. Some communications environments require an additional layer of control. NCR Comten answered this need by creating CDI, the Communications Device Interface. CDI provides connectionless, software-driven control over a communications interface. Much like NPI provides network communications without guarantee of end-to-end continuity, CDI provides point-to-point communications without any preconceptions of frame formats or end-to-end integrity assurances beyond the CCITT CRC checksums.

Gcom's CDI API provides access to this CDI interface, allowing an application to transmit raw data over the hardware. This extremely low-level interface can be useful in constructing specialized communications applications, or in dealing with unusual data-link protocols.

## *Using the CDI API*

Applications which employ CDI functionality have three phases. Initially the data stream must be opened and the API's facilities must be initialized. The second phase prepares the data stream for data transfer. Data transfer occurs in the third phase.

### **Preparing STREAMS**

The first phase conditions the CDI facility and opens a CDI datastream. The routines designed to support this include the following:

#### **cdi\_init(), cdi\_init\_FILE()**

Both these routines will initialize the global variables in the API and set the logging options.

#### **cdi\_set\_log\_size()**

This will set the maximum length of the logfile in bytes. If exceeded, the API will continue writing log information at the beginning of the file, overwriting old data with new as it cycles through the allotted space.

#### **cdi\_open\_data()**

This routine will open a data stream to the CDI driver.

Setting the log size and calling either `cdi_init_FILE()` or `cdi_init()` prepares the CDI interface for use; a call to `cdi_open_data()` will then provide a file descriptor (fid) for further operations.

## Manipulating the Device

The application's second phase of CDI communications prepares the opened stream or streams for use with a serial communication device. This phase involves attaching a stream to a CDI device and enabling the device for data transfer.

### **cdi\_attach\_req()**

When first opened, a data stream is in a disabled, unattached state. The API identifies this state with the `CD_UNATTACHED` define. To proceed in preparing the stream for data communication, this state must be advanced to the `CD_DISABLED` state. Two routines provide this functionality: *cdi\_attach\_req()* and *cdi\_put\_attach\_req()*. These routines will both associate the stream with a particular physical point of attachment (PPA). The PPA numbers are established in the configuration file. The numbers in the names of the `cdip.*` nodes correspond to the PPA numbers.

### **cdi\_wait\_ack()**

The difference between *cdi\_attach\_req()* and *cdi\_put\_attach\_req()* is that routines with “put” in the name are non-blocking in nature, meaning that they will return immediately and not wait on the CDI module's response. The *cdi\_wait\_ack()* routine provides a fairly straightforward mechanism for checking the CDI module's response after calling one of these non-blocking routines.

### **cdi\_detach\_req()**

Once the stream has been successfully attached, the application can call *cdi\_detach\_req()* to detach the PPA. The *cdi\_put\_detach\_req()* can also be employed, and *cdi\_put\_detach\_req()* will not wait for CDI to return an ACK or NACK response. Upon successful completion, the state is set to `CD_UNATTACHED`.

**cdi\_enable\_req()**

An attached stream must be enabled to be used for data transmission. The *cdi\_enable\_req()* and *cdi\_put\_enable\_req()* routines prepare the stream for data transmission. This moves the stream's state to CD\_ENABLED upon successful completion. If using the non-blocking version of the routine, *cdi\_put\_enable\_req()*, the stream will remain in the CD\_ENABLE\_PENDING state until the request is acknowledged.

**cdi\_disable\_req()**

Once enabled, a stream can be disabled again with either *cdi\_disable\_req()* or *cdi\_put\_disable\_req()*. Both routines will return a stream's state to CD\_DISABLED upon successful completion. If using the non-blocking version of the routine, *cdi\_put\_disable\_req()*, the stream will remain in the CD\_DISABLE\_PENDING state until the request is acknowledged.

**cdi\_allow\_input\_req()**

When a stream has been enabled, it can be used for data transmission. In order to allow receiving data, the application must call *cdi\_allow\_input\_req()* or *cdi\_put\_allow\_input\_req()*. This sets the state to CD\_INPUT\_ALLOWED. This indicates that the stream is in full-duplex operating mode. The routine should not be called for half-duplex operation.

## Data Communications

The third phase of communications with CDI is data transfer. After connecting to the stream, and connecting the stream to the device, and enabling input, the application can use the device to communicate. Data communications is implemented primarily in the first three of the following routines:

### **cdi\_write\_data()**

The bulk of data communications can be done using calls to *cdi\_write\_data()*.

### **cdi\_read\_data()**

Incoming data can be read with *cdi\_read\_data()*. Control information will be placed in the global variable *cdi\_ctl\_buf* and the global *cdi\_ctl\_cnt* will be set to indicate the size of the data in *cdi\_ctl\_buf*.

### **cdi\_rcv\_msg()**

Like *cdi\_read\_data()*, *cdi\_rcv\_msg()* retrieves data from a stream. *cdi\_rcv\_msg()* also has an optional 'flags' parameter.

Additional routines provide alternative means of handling data transfer for more specialized scenarios:

### **cdi\_put\_proto()**

Puts the proto message in *cdi\_ctl\_buf* onto the line. Does not wait for CDI's acknowledgement.

**cdi\_put\_data()**

Puts a buffer of data onto the line. Does not wait for the data to be acknowledged before returning.

**cdi\_put\_both()**

Puts both a buffer of data and the contents of *cdi\_ctl\_buf* on the line. Does not wait for the data to be acknowledged before returning.

**cdi\_put\_frame()**

This routine provides a simple technique for performing SDLC-style frame-level communications, albeit without providing the additional frame-sequencing and error correction that a full data link layer protocol would provide.

## Utility Routines

An additional set of routines stands apart from the main three phases, being at least marginally useful in multiple states. These routines are used for analysis and debugging the communications session. These routines include:

### **cdi\_modem\_sig\_req()**

This routine provides a means of setting modem signals.

### **cdi\_printf()**

This routine prints to the CDI API log using standard *printf()* conventions.

### **cdi\_decode\_modem\_sigs()**

This routine decodes modem signals and returns a descriptive string.

### **cdi\_decode\_ctl()**

This routine decodes the contents of *cdi\_ctl\_buf* and prints it to the logfile.

### **cdi\_print\_msg()**

This routine prints out a CDI message in hexadecimal notation.

### **cdi\_perror()**

Much like its namesake *perror(3)*, the *cdi\_perror()* routine prints an error message. The *cdi\_perror()* routine also prints to the API's log.

These routines provide some extended capability to a CDI application, including multiple printing and decoding routines as well as the capacity to instruct the modem to assert a specific set of modem signals.

## Global Variables

The Gcom CDI API uses global variables to provide/get information to/from the application. These globals are:

```
int          cdi_data_cnt ;
int          cdi_ctl_cnt ;
unsigned char cdi_data_buf[CDI_DATA_BUF_SIZE] ;
unsigned char cdi_ctl_buf [CDI_CTL_BUF_SIZE] ;
```

If an API routine modifies any of these variables, the routine descriptions below describe what is modified.

## Decoding Control Messages from the CDI Provider

From time to time the CDI provider will send control messages to the application. These messages should be decoded and if needed, responded to. The `<gcom/cdi.h>` file contains the definitions for these control messages and their data structures. Normally these control messages are placed in the `cdi_ctl_buf` global variable. See appendix A for a sample control message decoder routine.



# 2

---

## *API Reference*

This section of the manual provides an alphabetical reference guide to each of the API's functions.



## *cdi\_allow\_input\_req()*

**Prototype:** `int cdi_allow_input_req(int fid, int *state_ptr);`

**Parameters:** *fid*: file descriptor associated with data stream  
*state\_ptr*: will be filled with an integer representing the API state if provided

**Return Values:** 1: request sent, ACK received  
 0: request sent, NAK received  
 <0: error condition

**Include File(s):** <gcom/cdiapi.h>

**Description:** This routine sends an CD\_ALLOW\_INPUT\_REQ to *fid* and waits for a response. If successful, the CD\_ALLOW\_INPUT\_REQ will result in a transition to the CD\_INPUT\_ALLOWED state.

If *state\_ptr* is not NULL, the variable to which it points will be modified to represent the current state.

The values for the state are defined in <gcom/cdi.h>

Table 1 CDI Device States

| <i>#define for state</i> | <i>description</i>                                 |
|--------------------------|--|
| CD_UNATTACHED            | No PPA attached                                    |
| CD_UNUSABLE              | PPA cannot be used                                 |
| CD_DISABLED              | PPA attached                                       |
| CD_ENABLE_PENDING        | Waiting ACK of enable req                          |
| CD_ENABLED               | Awaiting use                                       |
| CD_READ_ACTIVE           | Input section enabled; disabled after data arrives |
| CD_INPUT_ALLOWED         | Input section permanently enabled                  |
| CD_DISABLE_PENDING       | Waiting ACK of disable req                         |
| CD_OUTPUT_ACTIVE         | Output section active only                         |

## *cdi\_attach\_req()*

**Prototype:** `int cdi_attach_req(int fid, long ppa, int *state_ptr);`

**Parameters:**

- fid*: file descriptor associated with data stream
- ppa*: the ppa to attach to
- state\_ptr*: will be filled with an integer representing the API state if provided

**Return Values:**

- 1: request sent, ACK received
- 0: request sent, NAK received
- <0: error condition

**Include File(s):** `<gcom/cdiapi.h>`

**Description:** This routine sends an CD\_ATTACH\_REQ to *fid* for *ppa* and waits for a response. The CD\_ATTACH\_REQ uses *ppa* to select a particular physical device for the data stream to communicate over.

If *state\_ptr* is not NULL, the variable to which it points will be modified to represent the current state.

The values for the state are defined in `<gcom/cdi.h>`

Table 2 CDI Device States

| <i>#define for state</i> | <i>description</i>                                 |
|--------------------------|--|
| CD_UNATTACHED            | No PPA attached                                    |
| CD_UNUSABLE              | PPA cannot be used                                 |
| CD_DISABLED              | PPA attached                                       |
| CD_ENABLE_PENDING        | Waiting ACK of enable req                          |
| CD_ENABLED               | Awaiting use                                       |
| CD_READ_ACTIVE           | Input section enabled; disabled after data arrives |
| CD_INPUT_ALLOWED         | Input section permanently enabled                  |
| CD_DISABLE_PENDING       | Waiting ACK of disable req                         |
| CD_OUTPUT_ACTIVE         | Output section active only                         |

## *cdi\_decode\_ctl()*

**Prototype:** void cdi\_decode\_ctl(char \*p);

**Parameters:** *p*: string to prepend to message

**Return Values:** NONE

**Include File(s):** <gcom/cdiapi.h>

**Description:** Decode the CDI protocol message contained in the global *cdi\_ctl\_buf* and print to the cdi log file pre-pending *p* to the entry.

## *cdi\_decode\_modem\_sigs()*

**Prototype:** `char *cdi_decode_modem_sigs(unsigned sigs);`

**Parameters:** *sigs*: modem signals

**Return Values:** A NULL terminated string

**Include File(s):** `<gcom/cdiapi.h>`

**Description:** Decode the modem signals in *sigs* and return a pointer to a descriptive string.

## *cdi\_detach\_req()*

**Prototype:** `int cdi_detach_req(int fid, int *state_ptr);`

**Parameters:** *fid*: file descriptor associated with data stream

*state\_ptr*: pointer to an integer to store state data in

**Return Values:** 1: request sent, ACK received

0: request sent, NAK received

<0: error condition

**Include File(s):** `<gcom/cdiapi.h>`

**Description:** Detaches *fid* from a particular physical point of attachment. This routine waits for a response.

If *state\_ptr* is not NULL, the variable to which it points will be modified to represent the current state.

The values for the state are defined in `<gcom/cdi.h>`.

Table 3 CDI Device States

| <i>#define for state</i> | <i>description</i>                                 |
|--------------------------|--|
| CD_UNATTACHED            | No PPA attached                                    |
| CD_UNUSABLE              | PPA cannot be used                                 |
| CD_DISABLED              | PPA attached                                       |
| CD_ENABLE_PENDING        | Waiting ACK of enable req                          |
| CD_ENABLED               | Awaiting use                                       |
| CD_READ_ACTIVE           | Input section enabled; disabled after data arrives |
| CD_INPUT_ALLOWED         | Input section permanently enabled                  |
| CD_DISABLE_PENDING       | Waiting ACK of disable req                         |
| CD_OUTPUT_ACTIVE         | Output section active only                         |

## *cdi\_disable\_req()*

**Prototype:** `int cdi_disable_req(int fid, unsigned long disposal,  
int *state_ptr);`

**Parameters:** *fid*: file descriptor associated with data stream  
*disposal*: what to do with queued data  
*state\_ptr*: will be filled with an integer representing the API state

**Return Values:** 1: request sent, ACK received  
0: request sent, NAK received  
<0: error condition

**Include File(s):** <gcom/cdiapi.h>

<gcom/cdi.h>

**Description:** Send a CD\_DISABLE\_REQ to the CDI provider and wait for a CD\_DISABLE\_CON or NAK. Once a disable is accepted, data may not be sent on *fid*. If data is queued, disposal can be set to CD\_FLUSH to discard undelivered data, CD\_WAIT to attempt to deliver unsent data, or CD\_DELIVER to deliver data prior to disabling the line.

If *state\_ptr* is not NULL, the variable to which it points will be modified to represent the current state.

Table 4 CDI Device States

| <i>#define for state</i> | <i>description</i>                                 |
|--------------------------|--|
| CD_UNATTACHED            | No PPA attached                                    |
| CD_UNUSABLE              | PPA cannot be used                                 |
| CD_DISABLED              | PPA attached                                       |
| CD_ENABLE_PENDING        | Waiting ACK of enable req                          |
| CD_ENABLED               | Awaiting use                                       |
| CD_READ_ACTIVE           | Input section enabled; disabled after data arrives |
| CD_INPUT_ALLOWED         | Input section permanently enabled                  |
| CD_DISABLE_PENDING       | Waiting ACK of disable req                         |
| CD_OUTPUT_ACTIVE         | Output section active only                         |

## *cdi\_enable\_req()*

**Prototype:** `int cdi_enable_req(int fid, int *state_ptr);`

**Parameters:** *fid*: file descriptor associated with data stream

*state\_ptr*: pointer to an integer to store state data in

**Return Values:** 1: request sent, ACK received

0: request sent, NAK received

<0: error condition

**Include File(s):** <gcom/cdiapi.h>

**Description:** This routine sends an CD\_ENABLE\_REQ to the CDI provider and waits for an CD\_ENABLE\_CON or NAK. If the CD\_AUTO\_ALLOW option has been used (as is the default), the stream is ready for data transfer once it has been enabled. If the CD\_AUTO\_ALLOW option is not being used, the application must call *cdi\_enable\_input()* to place the stream into full-duplex operational mode.

If *state\_ptr* is not NULL, the variable to which it points will be modified to represent the current state.

The values for the state are defined in <gcom/cdi.h>.

Table 5 CDI Device States

| <i>#define for state</i> | <i>description</i>                                 |
|--------------------------|--|
| CD_UNATTACHED            | No PPA attached                                    |
| CD_UNUSABLE              | PPA cannot be used                                 |
| CD_DISABLED              | PPA attached                                       |
| CD_ENABLE_PENDING        | Waiting ACK of enable req                          |
| CD_ENABLED               | Awaiting use                                       |
| CD_READ_ACTIVE           | Input section enabled; disabled after data arrives |
| CD_INPUT_ALLOWED         | Input section permanently enabled                  |
| CD_DISABLE_PENDING       | Waiting ACK of disable req                         |
| CD_OUTPUT_ACTIVE         | Output section active only                         |

## *cdi\_get\_a\_msg*

**Prototype:**

```
int cdi_get_a_msg(int    fid,
                  char   *buf,
                  int    size);
```

**Parameters:** *fid*: file descriptor associated with data stream

*buf*: points to a user-supplied buffer

*buf\_size*: length of memory pointed to by buffer

**Return Values:** >0: return code from the *getmsg()* call; generally indicates an undersized buffer

0: success

<0: error condition

**Include File(s):** <gcom/cdiapi.h>

**Description:** This routine is primarily used internally, but can be used to get one message from CDI. Any *M\_DATA* is placed in the caller's buffer, while the *M\_PROTO* (if any) is read into the global *cdi\_ctl\_buf*. The lengths of read data and read control information are placed in *cdi\_data\_cnt* and *cdi\_ctl\_cnt* respectively.

## *cdi\_get\_modem\_sigs*

**Prototype:** `int cdi_get_modem_sigs(int fid, int flag);`

**Parameters:** *fid*: file descriptor associated with data stream  
*flag*: passed to *cdi\_rcv\_msg()*; user can specify additional conditions on which the routine should return – see “*cdi\_rcv\_msg()*” on page 45 for valid values; 0 will return only on modem signal indication.

**Return Values:** >0: success, low 8 bits are a modem signal bitmask; suitable for passing to *cdi\_decode\_modem\_sigs()*  
 0: some other protocol message allowed by *flag* was received  
 <0: error condition

**Include File(s):** <gcom/cdiapi.h>

**Description:** This routine sends a modem signal poll to the device, then blocks until it receives either a `CD_MODEM_SIG_IND` or some other protocol message allowed in *flag*. If the routine is able to retrieve modem signals, they are returned as the low 8 bits in the return value. If some other protocol message is responsible for the routine’s return, the return value will be 0.

## *cdi\_init()*

**Prototype:** `int cdi_init(int log_optns, char *log_name);`

**Parameters:** *log\_optns:* a bitwise OR of the logging options (see below)  
*log\_name:* a NULL pointer or a pointer to a NULL-terminated string to use as the logfile name

**Return Values:** 1: success

**Include File(s):** `<gcom/cdiapi.h>`

**Description:** This routine sets up the initial environment for the API, opening a file named by the string at *log\_name* for use as the log file. If *log\_name* exists, it will be overwritten. If it doesn't exist, it will be created.

This procedure is designed so that it may be called multiple times with a NULL *log\_name* to alter the log options.

Table 6 Logging Options

| <i>#define for logging option</i> | <i>description</i>                               |
|-----------------------------------|--|
| CDI_LOG_FILE                      | log to file                                      |
| CDI_LOG_STDERR                    | log to stderr                                    |
| CDI_LOG_RX_PROTOS                 | log received M_PROTOS                            |
| CDI_LOG_TX_PROTOS                 | log transmitted M_PROTOS                         |
| CDI_LOG_ERRORS                    | log UNIX errors                                  |
| CDI_LOG_SIGNALS                   | log signal handling                              |
| CDI_LOG_RX_DATA                   | log received M_DATA                              |
| CDI_LOG_TX_DATA                   | log transmitted M_DATA                           |
| CDI_LOG_DISCARDS                  | see <code>cdi_rcv_msg ()</code>                  |
| CDI_LOG_VERBOSE                   | debug support                                    |
| CDI_LOG_DEFAULT                   | (CDI_LOG_FILE   CDI_LOG_STDERR   CDI_LOG_ERRORS) |

## *cdi\_init\_FILE()*

**Prototype:** `int cdi_init_FILE(int log_optns, FILE *filestream);`

**Parameters:** *log\_optns:* a bitwise OR of the logging options (see below)  
*filestream:* an already-opened filestream to use as the logfile or a NULL pointer

**Return Values:** 1: success

**Include File(s):** <gcom/cdiapi.h>

**Description:** This routine provides a means of passing the CDI API an already-opened file (filestream) for use as the CDI log file and to set or modify the logging options (log\_optns). Other than this, this routine is identical to cdi\_init() in that it sets up the initial environment for the API.

If filestream is NULL, only the logging options will be modified.

Table 7 Logging Options

| <i>#define for logging option</i> | <i>description</i>                               |
|-----------------------------------|--|
| CDI_LOG_FILE                      | log to file                                      |
| CDI_LOG_STDERR                    | log to stderr                                    |
| CDI_LOG_RX_PROTOS                 | log received M_PROTOS                            |
| CDI_LOG_TX_PROTOS                 | log transmitted M_PROTOS                         |
| CDI_LOG_ERRORS                    | log UNIX errors                                  |
| CDI_LOG_SIGNALS                   | log signal handling                              |
| CDI_LOG_RX_DATA                   | log received M_DATA                              |
| CDI_LOG_TX_DATA                   | log transmitted M_DATA                           |
| CDI_LOG_DISCARDS                  | see cdi_rcv_msg ()                               |
| CDI_LOG_VERBOSE                   | debug support                                    |
| CDI_LOG_DEFAULT                   | (CDI_LOG_FILE   CDI_LOG_STDERR   CDI_LOG_ERRORS) |

## *cdi\_modem\_sig\_poll*

**Prototype:** `int cdi_modem_sig_poll(int fid);`

**Parameters:** *fid*: file descriptor associated with data stream

**Return Values:** 0: request sent successfully  
<0: error condition

**Include File(s):** <gcom/cdiapi.h>, <gcom/cdi.h>

**Description:** This routine sends a modem signal poll to the device. The user is responsible for monitoring the stream for the response. A successful response will be in the form of a CD\_MODEM\_SIG\_IND. The or some other protocol message allowed in *flag*. If the routine is able to retrieve modem signals, they are returned as the low 8 bits in the return value. If some other protocol message is responsible for the routine's return, the return value will be 0.

## *cdi\_modem\_sig\_req()*

**Prototype:** `int cdi_modem_sig_req(int fid, unsigned sigs);`

**Parameters:** *fid*: file descriptor associated with data stream

*sigs*: requested modem signals

**Return Values:** 0: request sent successfully

<0: request failed

**Include File(s):** `<gcom/cdiapi.h>`

**Description:** Send a request to CDI provider on *fid* to set the modem signal to the value specified by *sigs*. When a modem signal changes, CDI will send a control message to the application with the primitive `CD_MODEM_SIG_IND`. The control packet is of type `cd_modem_sig_ind_t`, which is defined in `<gcom/cdi.h>`. See Appendix A for an example of a decoder which can be used to identify and examine packets like this.

This routine modifies the `cdi_ctl_buf` global variable.

## *cdi\_open\_data()*

**Prototype:** `int cdi_open_data(void);`

**Parameters:** None

**Return Values:** <0: error, errno should be appropriately set as for *open(2)*  
>=0: the fid of the new data stream

**Include File(s):** <gcom/cdiapi.h>

**Description:** This routine opens a CDI clone device and returns a descriptor associated with that data stream for the application to use.

## *cdi\_perror()*

**Prototype:** void cdi\_perror(char \*msg);

**Parameters:** *msg*: message to print

**Return Values:** None

**Include File(s):** <gcom/cdiapi.h>

**Description:** Similar to the UNIX *perror* routine, but prints to the CDI log file as well.

## *cdi\_printf()*

**Prototype:** `void cdi_printf(char *fmt, ...);`

**Parameters:** *fmt*: format string  
...: additional variable arguments for the format's substitutions

**Return Values:** None

**Include File(s):** `<gcom/cdiapi.h>`

**Description:** Refer to the *printf* man page for this routine. The length of the message to be printed must be less than 2000 bytes.

*cdi\_printf* performs an *fprintf()* to the log file.

## *cdi\_print\_msg()*

**Prototype:** void cdi\_print\_msg(unsigned char \*p, unsigned n,  
int indent);

**Parameters:** *p*: buffer containing data  
*n*: count of data  
*indent*: number of indent spaces in front of message

**Return Values:** None

**Include File(s):** <gcom/cdiapi.h>

**Description:** Print out a message in hexadecimal notation. This message will be indented by indent spaces.

## *cdi\_put\_allow\_input\_req()*

**Prototype:** `int cdi_put_allow_input_req(int fid);`

**Parameters:** *fid*: file descriptor associated with data stream

**Return Values:** 0: request sent  
<0: request NOT sent, error condition

**Include File(s):** <gcom/cdiapi.h>

**Description:** Send an CD\_ALLOW\_INPUT\_REQ on *fid* to the CDI provider and do not wait for a returned ACK or NAK. This routine modifies the *cdi\_ctl\_buf* global variable.

The application is responsible for determining the success or failure of the operation. See Appendix A starting on page 67 for an example of how to do this.

## *cdi\_put\_attach\_req()*

**Prototype:** `int cdi_put_attach_req(int fid, long ppa);`

**Parameters:** *fid*: file descriptor associated with data stream

*ppa*: the ppa to attach to

**Return Values:** 0: request successfully sent

<0: request not sent; error condition

**Include File(s):** <gcom/cdiapi.h>

**Description:** This routine prepares and sends fid an CD\_ATTACH\_REQ with a specified ppa and does not wait for a reply.

This routine modifies the *cdi\_ctl\_buf* global variable.

The application becomes responsible for determining the success or failure of the operation. See Appendix A starting on page 67 for an example of how to do this.

## *cdi\_put\_both()*

**Prototype:** `int cdi_put_both(int fid, char *header, int hdr_length,  
char *data_ptr, int data_length, int flags);`

**Parameters:**

- fid:* file descriptor associated with data stream
- header:* is a pointer to the control part of the message. May be NULL if *hdr\_length* is  $\leq 0$ .
- hdr\_length:* is the length of the control part of the message.
- data\_ptr:* is a pointer to the data part of the message. May be NULL if the *data\_length* is  $\leq 0$ .
- data\_length:* length of the data part of the message. One of *hdr\_length* and *data\_length* must be  $> 0$ . Both lengths can be  $> 0$ .
- flags:* determines retry

**Return Values:**

- 0: the messages were successfully sent to the CDI driver
- $< 0$ : the messages were not sent; error condition

**Include File(s):** `<gcom/cdiapi.h>`

**Description:** Send both control and data information on *fid*. If *flags* is set to `RetryOnSignal`, this routine will attempt to resend the data if interrupted by a signal.

See `<gcom/cdi.h>` for the message format definitions.

## *cdi\_put\_data()*

**Prototype:** `int cdi_put_data(int fid, char *data_ptr, int length,  
long flags);`

**Parameters:** *fid*: file descriptor associated with data stream

*data\_ptr*: buffer containing data

*length*: number of bytes to write

*flags*: retry flags

**Return Values:** 0: the message was successfully sent

<0: the message was not sent; error condition

**Include File(s):** <gcom/cdiapi.h>

**Description:** This routine writes *length* bytes from *data\_ptr* to *fid*. If *flags* are set to `RetryOnSignal`, this routine will attempt to resend the data if the send failed due to an `EINTR` error.

## *cdi\_put\_detach\_req()*

**Prototype:** `int cdi_put_detach_req(int fid);`

**Parameters:** *fid*: file descriptor associated with data stream

**Return Values:** 0: the message was successfully sent  
<0: the message was not sent; error condition

**Include File(s):** <gcom/cdiapi.h>

**Description:** Like *cdi\_detach\_req*, this routine detaches a data stream from a physical point of attachment (PPA). This routine works in a nonblocking mode, sending the request and returning immediately.

This routine modifies the *cdi\_ctl\_buf* global variable.

This routine will not wait for an ACK or NAK before returning. The success or failure of the actual send operation must be determined by the application. See Appendix A starting on page 67 for an example of how to do this.

## *cdi\_put\_disable\_req()*

**Prototype:** `int cdi_put_disable_req(int fid, unsigned long disposal);`

**Parameters:** *fid*: file descriptor associated with data stream

*disposal*: what to do with queued data

**Return Values:** 0: request successfully sent

<0: request NOT successfully sent; error condition

**Include File(s):** <gcom/cdiapi.h>

<gcom/cdi.h>

**Description:** Send a CD\_DISABLE\_REQ to the CDI provider and do not wait for a response. If data is queued, disposal can be set to CD\_FLUSH to discard undelivered data, CD\_WAIT to attempt to deliver unsent data, or CD\_DELIVER to deliver data prior to disabling the line.

This routine modifies the *cdi\_ctl\_buf* global variable.

This routine will not wait for an ACK or NAK before returning. The success or failure of the actual send operation must be determined by the application. See Appendix A starting on page 67 for an example of how to do this.

## *cdi\_put\_enable\_req()*

**Prototype:** `int cdi_put_enable_req(int fid);`

**Parameters:** *fid*: file descriptor associated with data stream

**Return Values:** 0: request successfully sent  
<0: request was NOT sent; error condition

**Include File(s):** <gcom/cdiapi.h>

**Description:** Transmits an CD\_ENABLE\_REQ to the CDI provider using *fid*. This routine does not wait for a response.

This routine modifies the *cdi\_ctl\_buf* global variable.

This routine will not wait for an ACK or NAK before returning. The success or failure of the actual send operation must be determined by the application. See Appendix A starting on page 67 for an example of how to do this.

## *cdi\_put\_frame()*

**Prototype:** `int cdi_put_frame(int fid, unsigned char address,  
                  unsigned char control, unsigned char *ptr,  
                  int count);`

**Parameters:** *fid*: file descriptor associated with data stream

*address*: address byte

*control*: control byte

*ptr*: buffer containing data

*count*: amount of data to send

**Return Values:** 0: frame was sent successfully

<0: frame was not sent successfully

**Include File(s):** <gcom/cdiapi.h>

**Description:** Send a buffer of data pre-pended with the *address* and *control* bytes. The *count* can be a maximum of 400.

## *cdi\_put\_proto()*

**Prototype:** `int cdi_put_proto(int fid, int length, long flags);`

**Parameters:** *fid*: file descriptor associated with data stream

*length*: number of bytes to write

*flags*: retry flags

**Return Values:** 0: the message contained in *cdi\_ctl\_buf* was successfully sent

<0: the message was not sent; error condition

**Include File(s):** `<gcom/cdiapi.h>`

**Description:** This routine writes the proto message built in the global *cdi\_ctl\_buf* to *fid*. If flags are set to `RetryOnSignal`, this routine will attempt to resend the data if the send failed due to an `EINTR` error.

## *cdi\_rcv\_msg()*

**Prototype:** `int cdi_rcv_msg(int fid, char *data_ptr, int bfr_len, long flags);`

**Parameters:** *fid*: file descriptor associated with data stream

*data\_ptr*: buffer for the received data

*bfr\_len*: max. amount of data to receive

*flags*: flags from options listed below

**Return Values:** >0: number of M\_DATA bytes received

0: A CDI control message is present at *cdi\_ctl\_buf*

<0: stream is no longer usable

**Include File(s):** <gcom/cdiapi.h>

**Description:** Read data from *fid* and place it into *data\_ptr*. If any control information is read, it is placed in the global *cdi\_ctl\_buf* with the length of the control information set in the global *cdi\_ctl\_cnt*.

Table 8 Flag values suitable for *flag* parameter to *cdi\_rcv\_msg()*

| <i>value</i> | <i>define</i>        | <i>meaning</i>   |
|--------------|----------------------|--|
| 0x001        | Return_error_ack     | Return if an error acknowledgement received                              |
| 0x002        | Return_info_ack      | Return if normal data acknowledgement received                           |
| 0x004        | Return_unidata_ack   | Return if unnumbered I-frame data acknowledgement received               |
| 0x008        | Return_error_ind     | Return if error indication received                                      |
| 0x010        | Return_disable_con   | Return if disable confirmation received                                  |
| 0x020        | Return_enable_con    | Return if enable confirmation received                                   |
| 0x040        | RetryOnSignal        | Retry operation if interrupted by signal while blocking (handle EINTR's) |
| 0x080        | Return_ok_ack        | Return if OK acknowledgement received                                    |
| 0x100        | Return_bad_frame_ind | Return if bad frame indication received                                  |
| 0x200        | Return_modem_sig_ind | Return if modem signal indication received                               |

## *cdi\_read\_data()*

**Prototype:** `int cdi_read_data(int cdi_data, char *buf, int cnt);`

**Parameters:** *cdi\_data*: file descriptor of the stream to receive the data from

*buf*: buffer for the received data

*cnt*: maximum amount of data to receive

**Return Values:** >0: number of M\_DATA bytes received

0: A CDI control message is present at *cdi\_ctl\_buf*

<0: stream is no longer usable

**Include File(s):** <gcom/cdiapi.h>

**Description:** This routine reads up to *cnt* bytes of data from *cdi\_data* into *buf*. If any control information is read, it is placed in the global *cdi\_ctl\_buf* with the length of the control information set in the global *cdi\_ctl\_cnt*.

## *cdi\_set\_log\_size()*

**Prototype:** `int cdi_set_log_size(long nbytes);`

**Parameters:** *nbytes*: the maximum size of the logfile

**Return Values:** 0: success

**Include File(s):** `<gcom/cdiapi.h>`

**Description:** This routine sets the maximum length of the log file in bytes. The log will be written in a circular fashion, returning to the beginning of the file and overwriting the oldest portions first. The ASCII string “End-of-log” will be written at the logical end of the log when the file is written in a circular fashion. If *nbytes*  $\leq$  0, the log size will not be limited and the log will be allowed to grow continuously.

## *cdi\_wait\_ack()*

**Prototype:** `int cdi_wait_ack(int fid, unsigned long primitive,  
int *state_ptr);`

**Parameters:**

- fid*: file descriptor associated with data stream
- primitive*: primitive to wait for
- state\_ptr*: pointer to an integer to store state data in

**Return Values:**

- 1: primitive ACK'd
- 0: primitive NAK'd
- <0: error condition

**Include File(s):** `<gcom/cdiapi.h>`

`<gcom/cdi.h>`

**Description:** This routine is called after sending a CD\_ATTACH\_REQ or CD\_ENABLE\_REQ on fid to wait for a returned confirmation or NAK. It compares primitive with that received in the message looking for a matching response. Any message received before the proper ACK/NAK will be discarded.

If *state\_ptr* is not NULL, the variable to which it points will be modified to represent the current state.

This routine modifies the *cdi\_data\_buf* and possibly the *cdi\_ctl\_buf* global variables. Refer to the *cdi\_get\_a\_msg()* section of this manual as this routine calls it to get the data and will modify some of the global variable.

## *cdi\_write\_data()*

**Prototype:** `int cdi_write_data(int cdi_data, char *buf, int cnt);`

**Parameters:** *cdi\_data*: file descriptor associated with data stream

*buf*: buffer containing the data

*cnt*: amount of data to send

**Return Values:** 0: the data was successfully sent

<0: the data was not sent; error condition

**Include File(s):** <gcom/cdiapi.h>

**Description:** This routine will write *cnt* bytes of data from *buf* to *cdi\_data*.

This routine modifies the *cdi\_data\_cnt* global variable.

## *cdi\_xray\_req()*

**Prototype:** `int cdi_xray_req( int fid,  
int upa,  
int on_off,  
int hi_wat,  
int lo_wat);`

**Parameters:** *fid:* The file descriptor of the data stream to use as the X-ray stream.

*upa:* The CDI UPA to be X-rayed.

*on\_off:* Non-zero to turn X-ray on, zero to turn X-ray off.

*hi\_wat:* A high-water mark for the stream. Should probably be relatively large (60000 or so).

*lo\_wat:* A low-water mark for the stream. Should probably be about 75% of the high-water mark.

**Return Values:** 0: The request was successfully sent.

<0: The request was not sent; error condition.

**Include File(s):** <gcom/cdiapi.h>

**Description:** This routine is used with a newly-opened data stream. When processed, the request sent by this routine links the stream to a CDI UPA. Any data traffic passing through the UPA will also be presented to this data stream. When data is presented to the X-ray stream, a descriptive header is prepended to the data, providing some insight into the nature of the data which follows. This header conforms to the format:

```
typedef struct
{
    cd_uns8    xr_type ;           /* 1 = send, 2 = receive */
    cd_uns8    xr_rsvd1 ;         /* reserved */
    cd_uns16   xr_seq ;           /* sequence number */
    cd_uns32   xr_timestamp ;     /* Rsys time (milliseconds) */
    cd_uns16   xr_modid ;         /* Rsys process number */
    cd_uns16   xr_upa ;           /* CDI UPA number */
} cd_x_ray_hdr_t ;
```

The Rsys time is the number of elapsed milliseconds since the Rsystem was initialized. The process number is an internal module ID number.

Sequence numbers can be useful in keeping track of flow control. Flow control for X-ray streams works a little differently than flow control for other streams. A “normal” stream will queue data until the high water mark is reached, then exert backpressure to stop data flow on that stream until the low water mark is reached. An X-ray stream begins discarding data when the high water mark is reached, then begins re-queuing it when the low water mark is reached. Monitoring sequence numbers will allow the user to determine when data was dropped.

The data which follow this header are the raw CDI message, formatted according to the protocol rules, with all the headers included. An application with some knowledge of the underlying protocols can decipher pieces of these messages to perform data tracing.

As a practical note, when X-ray streams are going to be used, the STREAMS configuration must have an extra CDI UPA or two beyond those needed to support the protocol drivers. These extra UPA’s are used by the X-ray streams.





















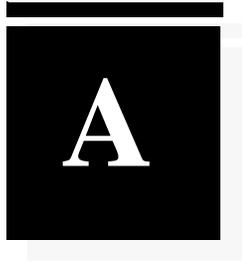












---

## *Sample Decoder*

The code sample below is a skeleton for a control message decoder. It is not complete, but should be used as a guide.

```
#include <gcom/cdiapi.h>
#include <gcom/cdi.h>

void
cdi_decode_ctl (prefix)
    char *prefix ;
{
    cd_ok_ack_t    *ap;                /* for primitive */

    if (prefix != NULL)
        pf("%s: ", prefix) ;
    ap = (cd_ok_ack_t *) cdi_ctl_buf ; /* proto ptr */

    switch ((int) ap -> cd_primitive)
    {
    case CD_INFO_ACK:                 /* Information ack */
        {
            cd_info_ack_t *ip;

            ip = (cd_info_ack_t *) ap;
            ...
            break;
        }

    case CD_ERROR_ACK: /* Error acknowledgement */
        {
            cd_error_ack_t *ep;
            ep = (cd_error_ack_t *) ap;
            ...
            break;
        }

    case CD_ENABLE_CON: /* Enable confirmation */
        {
            cd_enable_con_t *ep;

```

```

        ep = (cd_enable_con_t *) ap;
        ...
break;
    }

    case CD_DISABLE_CON: /* Disable confirmation */
    {
        cd_disable_con_t *dp;
        dp = (cd_disable_con_t *) ap;
        ...
        break;
    }

    case CD_ERROR_IND: /* Error indication */
    {
        cd_error_ind_t *ep;
        ...
        break;
    }

    case CD_UNITDATA_ACK: /* Data send acknowledgement */
    {
        cd_unitdata_ack_t *up;
        up = (cd_unitdata_ack_t *) ap;
        ...
        break;
    }

    case CD_UNITDATA_IND: /* Data receive indication */
    {
        cd_unitdata_ind_t *ip;
        ip = (cd_unitdata_ind_t *) ap;
        ...
        break;
    }

    case CD_BAD_FRAME_IND: /* Bad Frame */
    {
        cd_bad_frame_ind_t *pp;
        pp = (cd_bad_frame_ind_t *) ap;
        cdi_printf ("cd_bad_frame_ind: state 0x%lx \n", pp -> cd_state);

        switch (pp->cd_error)
        {
            case CD_FRMTOOLONG:
                cdi_printf ("Frame too long\n");
                break ;
            case CD_FRMNONOCTET:
                cdi_printf ("Frame not octet aligned\n");
                break ;
            case CD_EMPTY_BFR:
                cdi_printf ("Empty buffer\n");
                break ;
            case CD_BAD_CRC:
                cdi_printf ("Bad CRC\n");
                break ;
            case CD_FRM_ABORTED:
                cdi_printf ("Frame aborted\n");
                break ;
        }
    }

```

```

case CD_RCV_OVERRUN:
    cdi_printf ("Receiver overrun\n");
    break ;
default:
    cdi_printf ("cd_error 0x%x\n", pp -> cd_error);
    break ;
}
break;
}

case CD_MODEM_SIG_IND:                                     /* rcv modem signals */
{
    cd_modem_sig_ind_t *p;
    p = (cd_modem_sig_ind_t *) ap;
    if ( p->cd_sigs & CD_DTR )
    {
        /* DTR is asserted */
    }
    else
    {
        /* DTR is NOT asserted */
    }
    if ( p->cd_sigs & CD_RTS )
    {
        /* RTS is asserted */
    }
    else
    {
        /* RTS is NOT asserted */
    }
    if ( p->cd_sigs & CD_DSR )
    {
        /* DSR is asserted */
    }
    else
    {
        /* DSR is NOT asserted */
    }
    if ( p->cd_sigs & CD_DCD )
    {
        /* DCD is asserted */
    }
    else
    {
        /* DCD is NOT asserted */
    }
    if ( p->cd_sigs & CD_CTS )
    {
        /* CTS is asserted */
    }
    else
    {
        /* CTS is NOT asserted */
    }
    if ( p->cd_sigs & CD_RIN )
    {
        /* RING is asserted */
    }
    else

```

```

        {
            /* RING is NOT asserted */
        }
        cdi_printf ("cd_modem_sig_ind: %s\n",
                   cdi_decode_modem_sigs(p -> cd_sigs));
        break;
    }
    default:
    {
        cdi_printf ("cdi_decode_ctl: unknown primitive 0x%lx\n",
                   ap -> cd_primitive);
        break;
    }
} /* switch on type */
} /* cdi_decode_ctl */

/*
 * This is an excerpt from a program showing the decoder in use.
 * The example assumes that variables have been declared and initialized,
 * that the stream has been opened, and that communication is already
 * underway.
 * The purpose of the example is primarily to illustrate using
 * cdi_rcv_msg() to handle both normal data and control messages
 */

/* Any protocol message received should cause cdi_rcv_msg() to return
 * so we put all options into flag.
 */
flag = Return_error_ack | Return_info_ack | Return_unidata_ack |
       Return_error_ind | Return_disable_con | Return_enable_con |
       Return_ok_ack | Return_bad_frame_ind | Return_modem_sig_ind;

/* Adds RetryOnSignal so that an EINTR error will result in a retry */
flag |= RetryOnSignal;

/* Calls cdi_rcv_msg to receive incoming messages */
ret = cdi_rcv_msg(stream, buffer, length, flag);

/* In the event of data, process the data according to the application.
 * The routine to do this is not included in this example.
 */
if (ret > 0)
    cdi_process_buffer();

/* In the event of a control message, call our decoder, cdi_decode_ctl().
 * The sample decoder is provided in skeletal form above.
 */
if (ret == 0)
    cdi_decode_ctl(NULL);

/* In the event of a stream failure, call our exception handler.
 * This routine is not presented here, but would likely attempt to open
 * and initialize a new data stream, then begin re-enabling communication.
 */
if (ret < 0)
    cdi_handle_stream_error();

```