CDI

Application Program Interface Guide

February, 2006

Debra J. Schweiger, Geoffrey Gerrietts, Dave Grothe, Mikel Matthews, and Dave Healy were co-authors and subject matter experts for this manual.

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) 351-4241
FAX: (217) 351-4240
e-mail: support@gcom.com
homepage: http://gcom.com

CONTENTS

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.

New: cdi_open(NULL)
Old: 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.

The routine *dlpi_open_data* is equivalent to *dlpi_open(NULL*).

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.

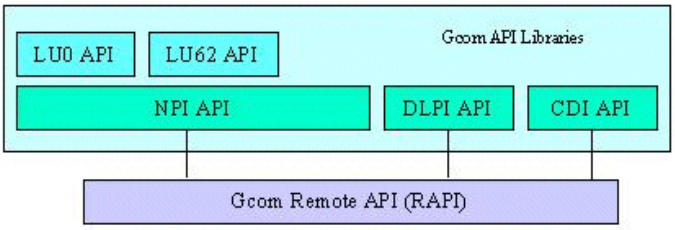
Gcom Remote API

GCOM'S Remote API (RAPI) is a library of functions that allows the standard GCOM APIs to operate on protocol stacks that are configured and running on a remote machine. It is especially useful in situations in which the application code resides on a server system and the protocol processing is performed on a GCOM Protocol appliance attached to the server via a LAN connection.

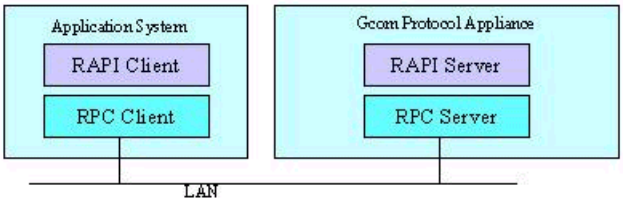
Architecture

The illustration, below, shows how the GCOM RAPI relates to all GCOM APIs. In the suite of GCOM API libraries, the NPI API interfaces to GCOM'S NPI driver for X.25, SNA and Bisync protocols. The GCOM DLPI interfaces to GCOM'S DLPI driver for link layer protocols such as LAPB, LAPD, HDLVC and Frame Relay. The GCOM CDI API library interfaces to GCOM'S synchronous protocol drivers directly for raw frame access.

.



Client Server Model



The Remote API is intended for use in a client/server environment. The user's application program, linked with the GCOM RAPI library, runs on the client system. The server system is typically a GCOM Protocol Appliance. It contains the communication hardware, protocol software and the Remote API server.

Running the RAPI Server

The GCOM Remote API server is named `Gcom_rapisvr`. It is usually unnecessary to run this program with any arguments. By default the program runs in the background. It can be run from the command line or from a shell script.

It is common to run `Gcom_rapisvr` under root from an "rc" script. However, if permissions are set appropriately on the files that are to be accessed remotely, it is perfectly possible to run `Gcom_rapisvr` from a non-root user id.

Note: Additional information on GCOM RAPI arguments, authentication, and other API routines can be found by accessing the GCOM RAPI white paper on the www.gcom.com web site.

Using the RAPI Library

In order to utilize the GCOM RAPI library it is necessary to link it into your program ahead of the "libgcom" library in order to link to the routines that perform the remote functions. A sample command line link for this is as follows:

```
cc -o foo foo.o /usr/lib/gcom/dlpiapi.a /usr/lib/gcom/rapi.a
/usr/lib/gcom/libgcom.a
```

Note: If RAPI library is omitted, then all file operations will be executed on the same machine on which the application program is running.

In the application program, be sure to use the correct API routine to open data streams on a remote system. The open routing of each of these routines is passed a parameter which is a pointer to a string which names the remote host. Passing a NULL pointer, or a pointer to an empty string, indicates that the file is to be opened on the local machine.

When opening or closing DLPI protocol data streams, use the functions:

Open routine: *cdi_open*
Close routine: *cdi_close*

Apart from using the specially provided open and close functions, there are no other programming interface considerations for making an application utilize remote protocol services.

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

#define for state	*description*
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

#define for state	*description*
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_close()

Prototype: `int cdi_close(int fd)`

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

Description: Closes the file that was opened by *cdi_open()*. This routine should be used to close the file rather than calling the system "close" routine directly. The *cdi_close* routine takes into account the possibility that the file may be opened to a remote server via the Remote API.

Return Values:

0	Success.
-1	Failure. Upon failure, "errno" is set to indicate the reason for failure.

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

#define for state	*description*
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_dial_req()

Prototype: `int cdi_dial_req(int fid, unsigned ppa, unsigned modem_sigs, char *dial_string, int dial_length);`

Parameters:

fid: file descriptor for an open CDI file

ppa: CDI UPA number of the port to operate on

modem_sigs: bit mask of modem signals to apply to the port

dial_string: bytes of dial data for modem

dial_length: number of bytes in *dial_string*

Return Values:

0: operation performed successfully

-1: operation failed due to signal but may succeed on retry

-2: operation failed and will likely fail on retry

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

<gcom.cdi.h>

Description: Applies the passed modem signals to the indicated physical port.

Intended use: Assert modem signals beneath a protocol stack to cause an auto-dial modem to:

- Dial out - Pass NULL for the *dial_string* parameter and zero for the *dial_length* parameter. (A future implementation may also pass the *dial_string* data to the port to cause an attached modem to dial out.)
- Hang up - Set the modem signals to zero or close the *fid* (which means the CDI file you open for the purpose of using this routine must remain open for the duration of the modem connection).

The operation of any protocol stack on *ppa* is unaffected by this routine unless the manipulation of modem signals causes line connection or disconnection.

The individual bit definitions for the modem signals reside in **<gcom/cdi.h>** as follows:

Table 4 Modem signal bit definitions

modem signal	*bit definition*
CD_DTR	0x01

Table 4 Modem signal bit definitions

modem signal	*bit definition*
CD_RTS	0x02
CD_DSR	0x04
CD_DCD	0x08
CD_CTS	0x10
CD_RI	0x20

Relationship to the *mdm_mask* parameter:

- The *modem_sigs* parameter is a bit mask of modem signals set into the port's *mdm_mask* parameter. Both output and input signals are valid in this mask. The output signals are asserted (or de-asserted if zero) and the input signals are sensed (or masked if zero).
- Some Gcom link layer protocols have a *mdm_msk* parameter that sets the sensed modem signals that must be active for the link layer to operate. Set the *mdm_msk* parameter and use the *cdi_dial_req* routine to produce the effect of bringing protocol stacks up and down along with the modem signals.

Relationship to the *cdi_open* routine:

- Use the *cdi_open* routine to obtain the *fid* parameter to pass to this routine.
- Call this routine after calling the *cdi_open* routine with no attach or enable functions performed on the *fid*. The operation is performed on the indicated *ppa* and does not require the attach step.

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 5 CDI Device States

#define for state	*description*
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 6 CDI Device States

#define for state	*description*
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 50 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 7 Logging Options

#define for logging option	*description*
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_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 8 Logging Options

#define for logging option	*description*
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:

```
Old: int cdi_open_data(void);
New: int cdi_open(char *hostname);
```

Parameters:

`hostname` A pointer to an ASCII string name of the machine on which the operation is to be performed. If the pointer is NULL, or points to an empty string, the operation is performed on the machine on which the call to the API library is made (i.e., the local host).

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.

The *cdi_open_data*(void) routine is equivalent to *cdi_open*(NULL).

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 57 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 57 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 header_length is <= 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 <= 0.
data_length:	length of the data part of the message. One of header_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 57 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 57 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 57 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 9 Flag values suitable for *flag* parameter to *cdi_rcv_msg()*

value	*define*	*meaning*
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* <= 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 (millisecs) */
   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.

A

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();
```