C111C C LIBRARY API ver. 3.00

Document Version:
2.0

Document Revision:
November, 29 2007 FP, UZ, MV

Description:
Helper function library for developing user applications to control a C111C Ethernet Camac Crate Controller.

Requirements:

Dependencies:

Referring documents:
[1] C111C USER'S MANUAL

Introduction

The C111C CAMAC Crate Controller can be controlled by a remote host by means of socket connections. Two different ports at the same TCP/IP address are used for exchanging commands and data between C111C and the host computer. Port 2000 is dedicated to *ASCII commands*, while port 2001 is dedicated to *Binary commands*.
All commands are available in ASCII form. An *ASCII command* is a text string that the host computer sends to C111C on port 2000. A complete reference of all ASCII commands can be found in the User's Manual, section "ASCII command reference". ASCII commands are easy to handle because they are simple text strings, but they can be quit slow for some applications. For this reason a command subset has been introduced, the *Binary commands* subset.
NOT all commands are available in binary form, but only those CAMAC commands for which speed can be an issue. Binary commands are faster, but more complex, because they use a special protocol, described in sections "TCP binary control socket" and "Binary commands reference" of the User's manual.

A control program can be written in any language, regardless of the platform and operating system running on the host computer. What is needed is just a piece of code to send and receive data on a socket connection. If binary commands are used, the programmer must also write some code to implement the binary protocol.

The purpose of the Library is to simplify and speed up code development for C/C++ programmers, hiding all details regarding socket connections and the binary protocol. No knowledge of socket programming and of the binary protocol is required and the user can avoid reading the manual's chapters about binary protocol.

The C Libray always uses binary commands. Three special commands (CMDS, CMDR, CMDSR) are available to send and receive commands and data in ASCII format.

Library files

The C Library is distributed in source code form (no precompiled binary).
The library consists in two files:
1) crate_lib.c
2) crate_lib.h

Compiling and Makefile

Using Linux platforms

The library files must be added to your project for use the library functions.

Example 1 (Application and library files are in the same folder)

1) Assuming that your source code application is in /home/jenet/jenet_appli.c
2) Assuming that the library source code application is in /home/jenet/jenet_appli.c
>From the shell prompt type:
> gcc jenet_appli.c crate_lib.c -o jenet_appli
This will build a jenet_appli application

If your application's source code is in a different folder you have to customize the
gcc command to reflect your settings.

Example 2 (Application and library files are in different folders)

1) Assuming that your source code application is in /home/jenet/jenet_appli.c
2) Assuming that the library source is in /home/jenet/library/crate_lib.c
3) Assuming that the working directory of the shell is /home/jenet
>From the shell prompt type:
/home/jenet> gcc jenet_appli.c /home/jenet/library/crate_lib.c -I/home/jenet/library/ -o jenet_appli

You can use the make tools and customize a Makefiles to build your project.

Example makefile:

```
#-*-makefile-*-
binaries = jenet_appli

all: compile

compile: $(binaries)

CC   = gcc
LINK = gcc

JENET_APPLI_SRC = crate_lib.c jenet_appli.c
JENET_APPLI_OBJ = $(JENET_APPLI_SRC:.c=.o)

jenet_appli: $(JENET_APPLI_OBJ) $(addsuffix .o, $(common))
$(LINK) -o $@ $^ -L. -lpthread

clean:
rm -f core *.o $(binaries) $(addsuffix .gdb, $(binaries))
```

Using Windows platforms

In the default software distribution there is a Workspace file (**crate_lib_win32.dsw**) created with Microsoft Visual Studio 6.0. The workspace includes some examples showing the correct use of the C Library.

Defines

```
//////////////////////////////////////////
//          Return Values
//////////////////////////////////////////

#define CRATE_OK                        0
#define CRATE_ERROR                     -1
#define CRATE_CONNECT_ERROR             -2
#define CRATE_IRQ_ERROR                 -3
#define CRATE_BIN_ERROR                 -4
#define CRATE_CMD_ERROR                 -5
#define CRATE_ID_ERROR                  -6
#define CRATE_MEMORY_ERROR              -7

//////////////////////////////////////////
//          BLK_TRANSF Opcode Defines
//////////////////////////////////////////

#define OP_BLKSS                        0x0
#define OP_BLKFS                        0x1
#define OP_BLKSR                        0x2
#define OP_BLKFR                        0x3
#define OP_BLKSA                        0x4
#define OP_BLKFA                        0x5

//////////////////////////////////////////
//          IRQ Type Defines
//////////////////////////////////////////

#define LAM_INT                         0x1
#define COMBO_INT                       0x2
#define DEFAULT_INT                     0x3


//////////////////////////////////////////
//          Miscellaneous
//////////////////////////////////////////

#define NO_BIN_RESPONSE                 0xA0
```

Configuration Functions

short CROPEN (char *address*);*

Description:

Opens a connection with the C111C Controller located at the IP address specified by the variable *address*.
This function performs an initialization of the library's internal variables and opens the socket connections available between the Host PC and the C111C Controller (ASCII, bin and irq).
If executed successfully, it returns a *crate identification number* (indicated in all functions as *crate_id*) that identifies, in a unique way, a single C111C Controller.
It is possible to control more than one C111C Controller at the same time, by simply calling this function multiple times for any of the C111C Controllers available over the LAN with the appropriate IP addresses.
NB: This function must be called before any other operation.

Parameters:

address – array of chars, it specifies the Ethernet IP address of the Controller, in the standard "dotted" form (i.e. xxx.yyy.www.zzz)

Results:

If executed successfully, it returns a crate_*id* that identifies, in a unique way, a single C111C Controller.
That crate_*id* must be used as first parameter in all of the other library functions;
In case of error, it returns one of the following error code:
CRATE_MEMORY_ERROR if the library cannot connect with another C111C Controller (maximum number of connection reached);
CRATE_CONNECT_ERROR, if no C111C controller responses at the specified ip address;
CRATE_BIN_ERROR, if an error occurs contacting the binary socket server of the specified C111C Controller;
CRATE_IRQ_ERROR, if an error occurs contacting the irq socket server of the specified C111C Controller;

Example:

```
short crate_id;
crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
        printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}
...
```

short CRCLOSE (short crate_id*);*

Description:

Close the connection with the C111C Controller identified by the parameter *crate_id* (see CROPEN).
NB: **This function must be called at the end of user application**, for all connected controllers.

Parameters:

crate_id – integer 16 bit, it specifies the C111C Controller (this value is returned by the function CROPEN)

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller cannot be disconnected in this moment;

Example:

```
short res, crate_id;
crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
      printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}
res = CRCLOSE(crate_id);
if (res < 0) {
      printf("Error %d closing connection with CAMAC Controller \n", crate_id);}
```

short CRIRQ (short crate_id*, IRQ_CALLBACK* irq_callback*);*

Description:

Allows to register a C function callback called by the library every time an IRQ event occurs on the C111C Controller specified by the parameter *crate_id.*
The IRQ_CALLBACK is defined as follows:

typedef void (*IRQ_CALLBACK) (short crate_id, short irq_type, unsigned int irq_value);

The first parameter of the C callback is the crate id; the second parameter is the irq type and the third is the irq value (its meaning is related to the irq type)
Irq type may be one of the following:

LAM_INT, if a LAM interrupt occurs on the C111C Controller, in this case the irq value is a 24-bit hexadecimal mask in which any bit set to 1 indicates the slot of the card that has generated a Lam irq (i.e irq_value = 0x100 means LAM irq from card in slot 9).
COMBO_INT, if an interrupt from combo occurs on the C111C Controller, in this case the irq value is a 4-bit hexadecimal mask in which:

bit 0 set to 1 indicates irq from combo 1,
bit 1 set to 1 indicates irq from combo 2,
bit 2 set to 1 indicates irq from dead time counter of combo 1,
bit 3 set to 1 indicates irq from dead time counter of combo 2;

DEFAULT_INT, if the default button was pressed on the C111C Controller, in this case the irq value must be ignored.

Parameters:

crate_id – integer 16 bit, it specifies the C111C Controller (see function CROPEN)
irq_callback – function callback, it defines a C callback function, called by the library when an IRQ event is catched by C111C Controller.

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified C111C controller is disconnected in this moment;

Example:

```
void IRQHandler(short crate_id, short irq_type, unsigned int irq_data)
{
	switch (irq_type) {
		case LAM_INT:
			// Do something when a LAM event occurs
			// Write your code here
			// …
			LACK(crate_id)
			break;
		case COMBO_INT:
			// Do something when a COMBO event occurs
			// Write your code here
			// …
			break;
		case DEFAULT_INT:
			// Do something when the "DEFAULT" button is pressed
			// Write your code here
			// …
			break;
	}
	return;
}

int main()
{
        short res, crate_id;
        crate_id = CROPEN("192.168.0.98");
        if (crate_id < 0) {
                printf("Error %d opening connection with CAMAC Controller \n", crate_id);}
        res = CRIRQ(crate_id, IRQHandler);
        if (res < 0) {
                printf("Error occurs registering callback: %d\n", res);
        }
}
```

short CRGET(short crate_id, CRATE_INFO *cr_info);

Description:

Allows to get some functional configuration parameters about the connection between the host and the C111C Controller identified by the parameter *crate_id* (see CROPEN).
NB: this function should be used only by expert users

The CRATE_INFO data struct is defined as follows:

```
typedef struct {
        char                    connected;

        int                     sock_ascii;
        int                     sock_bin;
        int                     sock_irq;

        short                   no_bin_resp;
        char                    tout_mode;
        unsigned int            tout_ticks;

        IRQ_CALLBACK            irq_callback;
        pthread_t               irq_tid;
} CRATE_INFO;
```

connected – byte, it is 1 if controller is successfully connected, otherwise it is set to 0.
sock_ascii – integer 32 bit, is the ASCII socket handle (**only for advanced users**).
sock_bin – integer 32 bit, is the binary socket handle (**only for advanced users**).
sock_irq – integer 32 bit, is the irq socket handle (**only for advanced users**).
no_bin_resp – integer 16 bit, used for sending binary commands without acknowledge from Controller (this option improve performance when sending relevant block of data) (**only for advanced users**).
tout_mode – byte, specifies the timeout mode (**actually is not used by the library**).
tout_ticks – integer 32 bit, this value defines the maximum interval the system waits before going in a timeout status (expressed in ms).
irq_callback – function callback, it defines a C callback function, called by the library when an IRQ event is catched by Controller handle (**only for advanced users**).
irq_tid – integer 32 bit, is the internal callback thread handle (**only for advanced users**).

Parameters:

crate_id – integer 16 bit, it specifies the C111C Controller (see function CROPEN)
cr_info – Pointer to CRATE_INFO struct

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified C111C controller is disconnected in this moment;

Example:

```
short res, crate_id;
CR_INFO cr_info;
crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
        printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}
res = CRGET(crate_id, &cr_info);
if (res < 0) {
      printf("Error occurs getting CRATE info: %d\n", res);
}
```

short CRSET(short **crate_id***, CRATE_INFO* ***cr_info***);*

Description:

Allows to set some functional configuration parameters about the connection between the host and the C111C Controller identified by the parameter *crate_id* (see CROPEN).
NB must be used only by expert user

For the CRATE_INFO structure, see function CRGET.

Parameters:

crate_id – integer 16 bit, it specifies the C111C Controller (see function CROPEN)
cr_info – Pointer to CRATE_INFO struct

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, crate_id;
CR_INFO cr_info;
crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
      printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}
res = CRGET(crate_id, &cr_info);
if (res < 0) {
      printf("Error occurs getting CRATE info: %d\n", res);
}
cr_info. tout_ticks = 100000;
res = CRSET(crate_id, &cr_info);
if (res < 0) {
      printf("Error occurs setting CRATE info: %d\n", res);
}
```

short CRTOUT(short **crate_id***, unsigned int* **tout***);*

Description:

Allows to set the maximum interval the system waits before go in a timeout status (expressed in ms) and consider the current operation aborted.

Parameters:

crate_id – integer 16 bit, it specifies the C111C Controller (see function CROPEN)
tout – integer 32 bit, this value defines the maximum interval the system waits before go in a timeout status (expressed in ms).

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, crate_id;
CR_INFO cr_info;
crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
       printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}
res = CRTOUT(crate_id, 100000);
if (res < 0) {
       printf("Error occurs setting timeout: %d\n", res);
}
```

short CBINR(short crate_id*, short* enable_resp*);*

Description:

Allows to enable/disable the response acknowledge when a binary command is sent.
Disabling response improves the performance, but this is a no reliable way to send commands.
NB must be used only by expert user

Parameters:

crate_id – integer 16 bit, it specifies the C111C Controller (see function CROPEN)
enable_resp – integer 16 bit, use NO_BIN_RESPONSE value (0xa0) to disable response over binary command socket, any other value enables the response.

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, crate_id;
CR_INFO cr_info;
crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
       printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}
res = CBINR(crate_id, 0xA0);
if (res < 0) {
       printf("Error occurs disabling response: %d\n", res);
}
```

short CSCAN(short* crate_id*, unsigned int* *scan_res*)

Description:
Performs a scan of the crate and returns a 24-bit hexadecimal mask in which any bit set to 1 indicates that the correspondent slot of the crate is filled with a card.

Parameters:
crate_id - integer 16 bit, it specifies the Controller (see function CROPEN)
scan_res - pointer to unsigned integer 32 bit, is a 24-bit hexadecimal mask in which any bit set to 1 indicates that correspondent slot of the crate is filled with a card.
(i.e scan_res = 0x101 means a card is present in slot 9 and slot 1).

Results:
If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong crate_id is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment.

Example:

```
short res, crate_id;
unsigned int scan_result;
crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
      printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}
res = CSCAN(crate_id, &scan_result);
if (res != CRATE_OK) {
      printf("Error occurs scanning the CRATE: %d\n", res);
}
for (i = 0; i < 24;i++) {
  if (scan_result & (1 << i)) {
    printf("The slot %d is filled with a card.\n", i + 1);
  }
}
```

ESONE functions

short CFSA(short crate_id, *CRATE_OP* *cr_op*);*

Description:

Executes a 24-bit CAMAC command; values of Q, X and DATA are returned in the CRATE_OP struct.
the CRATE_OP struct is defined as follows:

```
typedef struct {
        char F;
        char N;
        char A;
        char Q;
        char X;
        int  DATA;
} CRATE_OP;
```

F – byte, function identifier (accepted values: 0..27) (write only)
N – byte, slot identifier (accepted values: 1..24) (write only)
A – byte, address identifier (accepted values: 0..15) (write only)
Q – byte, status of Q line (read only)
X – byte, status of X line (read only)
DATA – integer 32 bit, data value (max 24-bit) (read/write)

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)
cr_op – pointer to a CRATE_OP struct, the following items in the CRATE_OP struct must be set before calling this function:
F,N,A;
If the function specified by the item F is a write operation also item DATA must be set before calling this function.;
If the function specified by the item F is a read operation then the item DATA will be updated with the 24-bit value returned from the card specified by the item N.
The Q and X status bytes are always set by the controller.

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, crate_id;
CRATE_OP cr_op;

crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
       printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}

cr_op.F = 17;
cr_op.N = 6;
cr_op.A = 0;
cr_op.DATA = 0x3F0000;

res = CFSA(crate_id, &cr_op);
if (res < 0) {
       printf("Error executing CFSA operation: %d\n", res);
}
```

short CSSA(short **crate_id*****, CRATE_OP*** ***cr_op*****);***

Description:

Executes a 16-bit CAMAC command; values of Q, X and DATA are returned in the CRATE_OP struct.
For the CRATE_OP struct see function CFSA:

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)
cr_op – pointer to a CRATE_OP struct, the following items in the CRATE_OP struct must be set before calling this function:
F,N,A;
If the function specified by the item F is a write operation also item DATA must be set before calling this function.;
If the function specified by the item F is a read operation then the item DATA will be updated with the 16-bit value returned from the card specified by the item N.
The Q and X status bytes are always set by the controller.

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, crate_id;
CRATE_OP cr_op;

crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
      printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}

cr_op.F = 17;
cr_op.N = 6;
cr_op.A = 0;
cr_op.DATA = 0x1234;

res = CSSA(crate_id, &cr_op);
if (res < 0) {
      printf("Error executing CSSA operation: %d\n", res);

}
```

short CCCZ(short **crate_id*****);***

Description:

Performs a Dataway init operation.

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, crate_id;

crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
      printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}

res = CCCZ(crate_id);
if (res < 0) {
      printf("Error executing CCCZ operation: %d\n", res);

}
```

short CCCC(short* crate_id*);

Description:

Performs a CRATE clear operation.

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, crate_id;

crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
      printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}

res = CCCC(crate_id);
if (res < 0) {
      printf("Error executing CCCC operation: %d\n", res);

}
```

short CTCI(short* crate_id*, char* *res*);

Description:

Performs a CAMAC Test Inhibit operation.

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)
res – pointer to byte, if executed successfully contains the result of operation (0 or 1)

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, res_op, crate_id;

crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
       printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}

res = CTCI(crate_id, &res_op);
if (res < 0) {
       printf("Error executing CTCI operation: %d\n", res);

}
printf("Test Inhibit results: %d\n", res_op);
```

short CCCI(short* crate_id*, char* data_in*);

Description:

Changes Dataway Inhibit to a specified value.

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)
data_in – byte, contains the new Dataway inhibit value (0 or 1)

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, res_op, crate_id;

crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
       printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}

res_op = 1
res = CCCI(crate_id, res_op);
if (res < 0) {
       printf("Error executing CCCI operation: %d\n", res);

}
```

short CTLM(short crate_id*, char* slot*, char* *res*);*

Description:

Performs a CAMAC test LAM on specified slot. If slot = -1, it checks for a LAM on any slot.

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)
slot – byte, slot identifier (1..23)
res – pointer to byte, if executed successfully contains the result of operation

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, res_op, crate_id;

crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
      printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}

res = CTLM(crate_id, 6, &res_op);
if (res < 0) {
      printf("Error executing CTLM operation: %d\n", res);

}
```

short CCLWT(short crate_id*, char* slot*);*

Description:

CAMAC waits for LAM event on specified slot; if N = -1, it waits for a LAM on any slot.

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)
slot – byte, slot identifier (1..23)

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res , crate_id;
crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
      printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}

res = CCLWT(crate_id, 6);
if (res < 0) {
      printf("Error executing CTLM operation: %d\n", res);
}
```

short LACK(short* crate_id*);

Description:

Performs a LAM acknowledge. Must be called in the IRQ Handler, see function CRIRQ.

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)
slot – byte, slot identifier (1..23)

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
void IRQHandler(short crate_id, short irq_type, unsigned int irq_data)
{
        switch (irq_type) {
                case LAM_INT:
                        // Do something when a LAM event occurs
                        // Write your code here
                        // …
                        LACK(crate_id)
                        break;
                case COMBO_INT:
                        // Do something when a COMBO event occurs
                        // Write your code here
                        // …
                        break;
                case DEFAULT_INT:
                        // Do something when the "DEFAULT" button is pressed
                        // Write your code here
                        // …
                        break;
        }
        return;
}

int main()
{
short res, crate_id;
crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
                printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}
res = CRIRQ(crate_id, IRQHandler);
if (res < 0) {
                printf("Error occurs registering callback: %d\n", res);
}
...
}
```

short CLMR(short **crate_id***, unsigned int* ***reg***);*

Description:

Returns current LAM register, in hex.

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)
reg – unsigned integer 32 bit, is a 24-bit hexadecimal mask in which any bit set to 1 indicates the slot of the card that has generated a LAM request (i.e reg = 0x100 means LAM from card in slot 9).

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, crate_id;
unsigned int reg;

crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
       printf("Error %d opening connection with CAMAC Controller \n", crate_id);}

res = CLMR(crate_id, &reg);
if (res < 0) {
       printf("Error executing CLMR operation: %d\n", res);
}
```

short CTSTAT(short **crate_id***, char* ***Q***, char* ***X***);*

Description:

Returns Q and X values (from last access on bus)

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)
Q – pointer to byte, status of Q line
X – pointer to byte, status of X line

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, crate_id;
char q, x;

crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
       printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}

res = CTSTAT(crate_id, &q, &x);
if (res < 0) {
       printf("Error executing CTSTAT operation: %d\n", res);
}
```

short NOSOS(short* crate_id*, char* nimo*, char* value*);

Description:

Performs a fast single NIM out operation.

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)
nimo – byte, NIM output to be modified
value – byte, value to be set on the NIM output

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, crate_id;


crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
       printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}

res = NOSOS(crate_id, 0, 1);  //NIM output 0 is set to 1
if (res < 0) {
       printf("Error executing NOSOS operation: %d\n", res);
}
```

Generic command functions

short **CMDS** *(short* crate_id*, char* *cmd*, int* size*);*

Description:

Sends a generic ASCII command to the ASCII command socket of the Controller identified by *crate_id*.
A complete list of all ASCII commands can be found in the User's Manual, Section "ASCII commands reference".
The command CMDS simply sends a command without reading the answer. Please note that the Camac controller always replies to any ASCII command, even if the answer is not read. This may cause timeout or TCP/IP read buffer overrun on some hosts depending on their configuration and operating system.
The reply can be read using the command CMDR. If there are no reasons for sending commands without reading the reply, the use of the command CMDSR is suggested.

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)
cmd – array of chars, defines the command to be sent
size – integer 32 bit, size of command (number of characters)

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameter;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, crate_id;


crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
       printf("Error %d opening connection with CAMAC Controller \n", crate_id);
}

res = CMDS(crate_id, "CFSA 17 6 0 1" , 13);
if (res < 0) {
       printf("Error executing CMDS operation: %d\n", res);
}
```

short **CMDR** *(short* crate_id*, char* *resp*, int* size*);*

Description:

Allows to read a response to a single command sent to the ASCII command socket of the Controller identified by *crate_id*.

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)
resp – array of chars, defines the buffer in which the response is stored
size – integer 32 bit, maximum number of bytes to be read

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, crate_id;
char response[32];

crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
      printf(“Error %d opening connection with CAMAC Controller \n”, crate_id);
}

res = CMDS(crate_id, "CFSA 17 6 0 1" , 13);
if (res < 0) {
      printf("Error executing CMDS operation: %d\n", res);
}

res = CMDR(crate_id, response, 32);
if (res < 0) {
      printf("Error executing CMDR operation: %d\n", res);
}
printf("Response: %s\n", response);
```

short CMDSR (short crate_id*, char* *cmd*, char* *resp*, int* size*);*

Description:

Allows to send a command to the ASCII command socket and read the response from the Controller identified by *crate_id.*
A complete list of all ASCII commands can be found in the User's Manual, section “ASCII commands reference”.

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)
cmd – array of chars, defines the command to be sent
resp – array of chars, defines the buffer in which the response is stored
size – integer 32 bit, maximum number of bytes to be read

Results:

If executed successfully, it returns CRATE_OK, otherwise returns one of the following errors:
CRATE_ID_ERROR, if a wrong *crate_id* is specified as parameters;
CRATE_CONNECT_ERROR, if the specified controller is disconnected in this moment;

Example:

```
short res, crate_id;
char cmd[32], response[32];

crate_id = CROPEN("192.168.0.98");
if (crate_id < 0) {
      printf(“Error %d opening connection with CAMAC Controller \n”, crate_id);
}

strcpy(cmd, "CFSA 17 6 0 1");

res = CMDSR(crate_id, cmd, response, 32);
if (res < 0) {
      printf("Error executing CMDSR operation: %d\n", res);
}

printf("Response: %s\n", response);
```

Block transfer functions

short BLKBUFFS (short crate_id, *short* value*);*

Description:

Set the current block transfer buffer size. This value affects the numbers of data-words transferred in a single TCP/IP transaction. A low value will cause a great number of TCP/IP transactions. Default value is 16.
A detailed description of Block Transfer operations can be found in [1].

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)

value: integer 16 bit – block transfer buffer size (allowed values: 1 to 256)

Results:

CRATE_OK: Operation completed successfully;
CRATE_ERROR: Operation failed

short BLKTRANSF (short crate_id, *BLK_TRANSF_INFO* *blk_info, *unsigned int* *buffer*);*

Description:

Performs a block transfer operation.
A detailed description of Block Transfer operations can be found in [1].

Parameters:

crate_id – integer 16 bit, it specifies the Controller (see function CROPEN)

blk_info - data structure:

- `opcode`: specify the type of block transfer; can be one of the following:
  - `OP_BLKSS` to perform a 16 bit block transfer STOP mode;
  - `OP_BLKFS` to perform a 24 bit block transfer STOP mode;
  - `OP_BLKSR` to perform a 16 bit block transfer REPEAT mode;
  - `OP_BLKFR` to perform a 24 bit block transfer REPEAT mode;
  - `OP_BLKSA` to perform a 16 bit block transfer ADDRESS SCAN mode;
  - `OP_BLKFA` to perform a 24 bit block transfer ADDRESS SCAN mode;
- `F`: function identifier (0..27)
- `N`: slot identifier (1..23) (specifies the start slot in ADDRESS SCAN mode)
- `A`: address identifier (0..15) (non significant in ADDRESS SCAN mode)
- `totsize`: total number of data words to be transferred
- `blksize`: the current block transfer buffer size
- `timeout`: timeout in seconds (significant only in ADDRESS SCAN mode)

buffer: array of integer 32 bit: if the blk_info.opcode specify a read operation, buffer will be filled with the data read during the block transfer operation. If the blk_info.opcode specify a write operation, buffer must be already filled with the data being transferred during the block transfer operation. The application must provide a pointer to a buffer referring a memory area with enough bytes allocated to perform the operation safely.

Results:

CRATE_OK: Operation completed successfully; in this case the blk_info.totsize will be filled with the actual data size effectively transferred by the CAMAC Controller
CRATE_ERROR: Operation failed

Example:

```
int main(int argc, char *argv[])
{
	short crate_id;
	int i, j;
	int resp;
	// Block transfer operation sends 16-bit data separated in lines
	 // This parameter sets the amount of data per line
	int block_data_size;

	int total_data_size; // This is the total 16-bit data to be sent

	char blk_ascii_buf[2048];
	unsigned int blk_transf_buf[300];

  BLK_TRANSF_INFO blk_info;
  CRATE_OP op;

	printf("Block Transfer Test\n");

/*
	=================================================
	Open connection with a Camac controller
	=================================================
*/
  printf("Initializing communication parameters...\n");

  crate_id = CROPEN("192.168.0.98");
  if (crate_id < 0) {
	printf("Error %d opening connection with CAMAC Controller \n", crate_id);
	return 0;
  }

/*
	=================================================
	Invoke Block transfer Q-stop mode
	(write operation on Crate Module N address 0)
	=================================================
*/
	block_data_size = 16;
	total_data_size = (block_data_size * 5); //Sent 80 16-bit data items

	printf("Start block transfer write\n");

	// Prepare send test pattern
	for (j = 0; j < 5; j++) {
		for (i = 0; i < block_data_size; i++) {
			if (i & 1)
				blk_transf_buf[i + (j * block_data_size)] = 0x5555;
			else
				blk_transf_buf[i + (j * block_data_size)] = 0xAAAA;
		}
	}

	blk_info.opcode = OP_BLKSS;
	blk_info.F = 16;
	blk_info.N = 6;
	blk_info.A = 0;
	blk_info.blksize = block_data_size;
	blk_info.totsize = total_data_size;
	blk_info.timeout = 0;

	resp = BLKTRANSF(crate_id, &blk_info, blk_transf_buf);

	if (resp != CRATE_OK) {
		printf("ERROR: Negative response from socket server\n");
		return 0;
	}

	printf("Total data written: %d\n", blk_info.totsize);

	printf("End of block transfer write\n");

/*
	====================================================
	Invoke Block transfer Q-stop mode
	(read operation on Crate Module N address 0)
	====================================================
*/

	block_data_size = 32;
	total_data_size = (block_data_size * 5);

	printf("Start block transfer read\n");

	blk_info.opcode = OP_BLKSS;
	blk_info.F = 0;
	blk_info.N = 6;
	blk_info.A = 0;
	blk_info.totsize = total_data_size;
	blk_info.blksize = block_data_size;
	blk_info.timeout = 0;

	resp = BLKTRANSF(crate_id, &blk_info, blk_transf_buf);
	if (resp != CRATE_OK) {
		printf("ERROR: Negative response from socket server\n");
		return 0;
	}

	printf("Total data read: %d\n", blk_info.totsize);
	for (i = 0; i < blk_info.totsize; i++) {
		// Show received buffer
		if ((i > 0) && ((i % 10) == 0)) {
			printf("\n");
		}
		printf("%06X ", blk_transf_buf[i]);
	}
	printf("\nEnd of block transfer read\n");

	return 0;
}
```