EDAC40: ethernet-connected 40-channel Digital-to-Analog converter module

User guide and service manual

January 22, 2013

Figure 1: EDAC40 Ethernet control unit

1 Specifications

The EDAC-40 unit is a network-enabled device which is intended to provide multi-channel voltage output remotely controlled by computer. It was originally designed to drive deformable mirrors produced by Flexible Optical B.V. Technical data are listed in Table 1.

Parameter	Value
Analog outputs	40
Output resolution	16 bits (65536 levels)
Maximum output span	12 V (adjustable with *gain* parameter)
Minimum output level	-12 V (adjustable with *offset* parameter)
Maximum output level	+12 V (adjustable with *offset* parameter)
Output mode	synchronous for all channels
Short-circuit current, each channel	15 mA
Capacitive load, each channel	$\leq$2200 pF
Network protocols for data	UDP, TCP (port 1234)
Minimal data packet size	8 bytes
Maximum data packet size	86 bytes
Network service protocols	DHCP client, ICMP
Automatic device recognition	Microchip *discover* protocol (UDP port 30303)
Method of range adjustment	software, stored in nonvolatile memory (NVRAM) in the unit
Supply power	+5 V DC

Table 1: EDAC-40 technical specifications

2 General design and principle of operation

The unit is designed as single PCB mounted in compact plastic enclosure. Its layout is shown schematically in Fig 2. The board carries two double-row angle 20-pin male connectors for analog outputs (J4: channels 1-20, J5:channels 21-40), network RJ-45 connector J1, power connector J3. Channel numbers are market on the rear panel of the device. A set of header jumpers JP1–JP4 provides the possibility to modify assignments of selected outputs. Jumper JP1 allows swapping of pins 1 and 2 (channels 1 and 2) of output connector J4. Jumper JP2 provides the same function for pins 1 and 2 of connector J5 (channels 21 and 22). Jumpers JP3 and JP4 connect either DAC output (2-3 position) or ground (1-2 position) to pins 1 of connectors J4 and J5 correspondingly. Possible settings variants are summarized in Table 2. To get access to external controls one should disengage lower and upper parts of the housing by simultaneously depressing of their side surfaces.

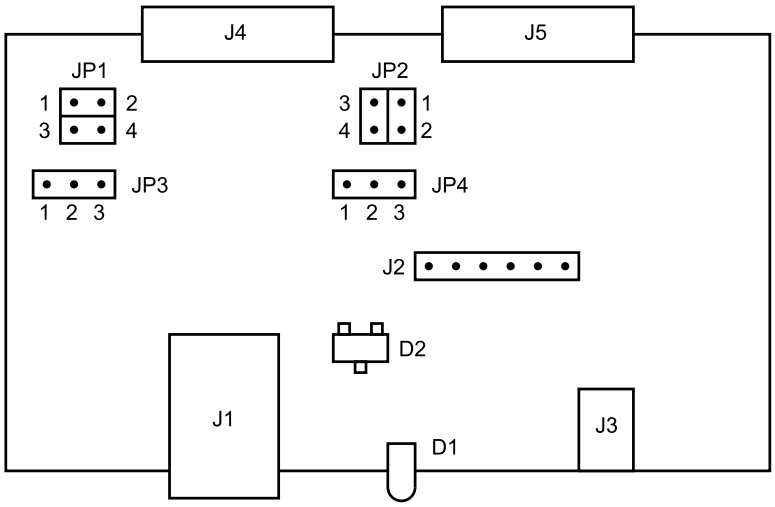

Figure 2: EDAC40 external connections and indication

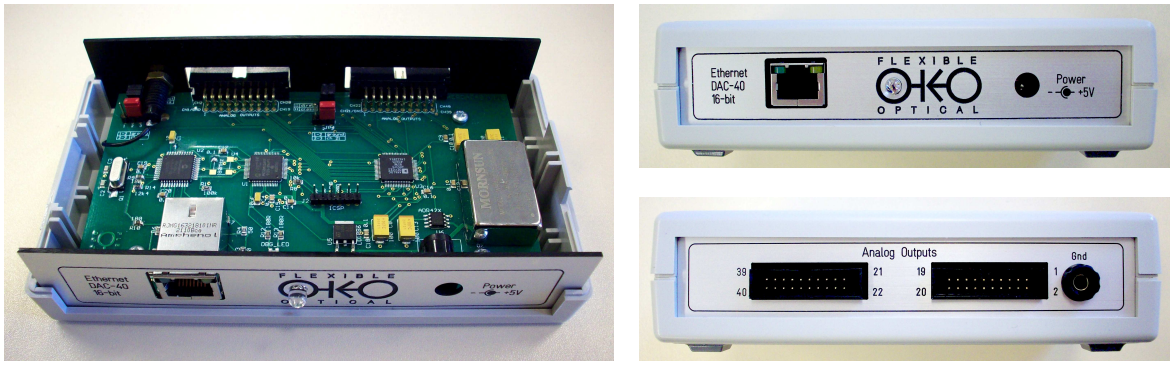

Figure 3: EDAC40 board view (left), front and rear panels (right)

JP1	JP3	Pin 1 of J4	Pin 2 of J4
direct	1–2	ground	channel 2
(1-2 and 3-4 shorted)	2–3	channel 1	channel 2
cross	1–2	channel 2	ground
(1-3 and 2-4 shorted)	2–3	channel 2	channel 1
JP2	JP4	Pin 1 of J5	Pin 2 of J5
direct	1–2	ground	channel 22
(1-2 and 3-4 shorted)	2–3	chaneel 21	channel 22
cross	1–2	channel 22	ground
(1-3 and 2-4 shorted)	2–3	channel 22	channel 21

Table 2: EDAC40 jumper settings

3 Data exchange protocol

The module expects data packages on port 1234 using either UDP (datagram) or TCP (stream-oriented) protocols. While data format for both protocols is exactly the same, using of each of them has some pro and contras. UDP is packet-oriented connection-less protocol. It is pretty fast and "unreliable" by design. Since there is no way to find out if the packet received by other side, packets in principle can get lost, received more than once or arrive out of order. TCP on the other hand implements intrinsic acknowledgment mechanism which guarantees data integrity. UDP protocol provides approximately ten times higher throughput and recommended for typical laboratory environment when all network devices connected to the same local network.

Total number of transmitted bytes may vary from 8 to 86 bytes per frame depending of number of addressed channels. Frame structure is illustrated by table 3. Possible function codes (byte 6) are listed in the table 4. If data for particular channel are present in the frame, corresponding mask bit (bytes 0 to 5) should be set to 1. Data register, gain and offset are independent for each of 40 channels. Global 14-bit offset applies to all channels. Output voltage (VOUT) can be calculated as follows:

$$\text{DAC CODE} = \text{INPUT CODE} \times (\text{GAIN} + 1) + \text{OFFSET} - 2^{15}$$

$$\text{VOUT} = 4 \times \text{VREF} \times \left(\frac{\text{DAC CODE}}{2^{16}} - \frac{\text{GLOBAL OFFSET CODE}}{2^{14}} \right)$$

where DAC CODE should be within the range of 0 to 65535, VREF $= 3.0V$ – voltage reference. GLOBAL OFFSET CODE is loaded to the offset DAC. Depending on the chosen offset value the output span could be bipolar, unipolar, symmetric and asymmetric. For instance, offset value of 0x1FFF (default) gives the output span of $-6 \ldots +6$V, value of 0 – span of $0 \ldots +12$V, value of 0x3FFF – span of $-12 \ldots 0$ and offset value of 0x1555 – span of $-4 \ldots +8$ volts.

Offset	Size (bytes)	Description
0	5	Channel mask
+5	1	Command code
+6	2×(number of channels addressed)	Data (16-bit unsigned integer per channel)

Table 3: EDAC40 data frame format

Code (binary)	Code (decimal)	Descrioption	Length (bits)	Default value
0000 0000	0	Set DAC value	16	
0000 0001	1	Set Offset	16	0x8000
0000 0010	2	Set Gain	16	0xFFFF
0000 0011	3	Set Offset DAC (global)	14	0x1FFF
0000 0100	4	Save current setting into NVRAM	N/A	N/A

Table 4: Function codes

The module acts as DHCP (Dynamic Host Configuration Protocol) client for automatic obtaining of its IP address. The module implements Microchip Announce/Discover protocol responding to discover request on UDP port 30303 with a string, containing device signature "EDAC40" and hardware (MAC) address.

4 Getting started

4.1 Connecting to a network

The device can be connected to network and configured in a number of different ways. It is an end user responsibility to configure their network and workstations properly. Two common cases are sketched briefly below.

- The EDAC40 unit is connected to a local network in the laboratory. It is possible to have many EDAC40 units connected to the same network. Physically they should be connected to some network switch or router using UTP-5 cable with either straight or cross wiring. The same network segment is used for traffic within the laboratory/office, addressing EDAC40 units and (possibly) access to the Internet. IP address should be provided by DHCP server running on either network router or one of the computers connected to the network. The machines within the network are typically automatically assigned private IP addresses in one of the ranges 10.0.0.0–10.255.255.255, 172.16.0.0–172.31.255.255 or 192.168.0.0–192.168.255.255 depending on the configuration. The network may be connected to outside world via router or switch. In such case NAT

(Network Address Translation) is usually used for Internet access. It is still possible to access individual local nodes ("servers"), including EDAC40 devices from outside. This can be accomplished with so called "port forwarding". Exact steps necessary for configuring varies for different router makes/models and are outside of the scope of this manual.

- The device is directly connected to a RJ-45 port of a LAN adapter of the PC with no other devices sharing the same segment of the network. The UTP-5 cable used for the connection may have either straight or cross wiring. The computer should be configured to obtain the IP automatically. As there is no DHCP server available in that case, both PC workstation and DAC unit fall back to use of "link-local" IP addresses within the range 169.254.0.0–169.254.255.255. The process of initial network self-configuration of a modern Windows PC typically completes within 15 seconds but may take as long as a few minutes in some cases. Accordingly to standards network packages in this case are not routed (i.e. they never leave that particular network segment). It is also possible (though not recommended) to have DHCP server running on the network interface to which the EDAC40 device is connected, in that case special care should be taken to avoid routing table misconfiguration. Other network interfaces of the same workstation can be assigned for other tasks, such as the Internet access. It is possible to connect several EDAC units to a single PC using different RJ-45 ports of one multiport LAN adapter, several LAN adapters or one adapter and a network switch/hub.

Technically it is perfectly possible to mix above mentioned configurations but the end user is expected to understand well what they are doing.

4.2 Testing and adjusting

Before using the device it is required to confirm jumper settings according to specifications of connected device(s) as described above. Testing and output range adjustment could be done as follows.

- Plug 5VDC power supply connector into J3.
- Connect a UTP5 patch cord of sufficient length between RJ-45 connector J1 of the unit and other network device, such as LAN card of a personal computer or network switching hub (see the section above for details. Yellow LED above the connector indicates existence of correct link.
- Run edac40gui program (provided on software CD). The connected device MAC address appears in the combo box in the upper part of the application window (Fig 4). If there are multiple EDAC40 devices in the network, the one under testing should be selected from the drop-down list. The application provides controls to set either desired constant level to all output channels or generate square wave or saw-like signal. Internal gain and offset settings can be programmed and finally stored into non-volatile memory (NVRAM).
- Connect an oscilloscope probe to one of the channels.
- Choose amplitude "1" (full scale), shape — "square" and press "Go". Square-shaped wave signal should appear on the screen of the scope. Low and high levels correspond to minimum (0) and maximum (65536) DAC codes.
- The output levels could be adjusted in two ways: by applying analog offset formed by separate on-chip DAC (Offset DAC controls, it is negative and global for all channels) and/or by changing digital codes for gain and offset (in principle, they could be set individually for every separate channel). Full set of adjustments is available using low-level network protocol or provided software library. Only global analog offset and gain adjustments are implemented in current version of edac40gui program. To use adjustment controls please check "Enable" check-box, then press "Restore defaults" button. Adjust gain and offset parameters with sliders of spin boxes to get desired output voltage range. Please note, that some combinations of Offset DAC, Gain and Offset parameters will lead to limiting of output by amplifier ("saturation").
- The user might want to set constant output level (select "Constant" radio button in the "Shape" group) and check the voltage with digital voltmeter for both zero and full scale codes.
- When satisfied with settings, it is advisable to save the parameters into non-volatile memory (NVRAM): press "Save to NVRAM" button.

Please note, that analog electrical connections between DAC unit(s), high-voltage amplifier(s) and a adaptive mirror should be performed with the devices powered off. Strong electromagnetic noise or electrostatic discharge can sometimes lead to temporary unit malfunction. In such rare cases device reset can be performed by cycling its power (unpluging/repluging of +5VDC supply).

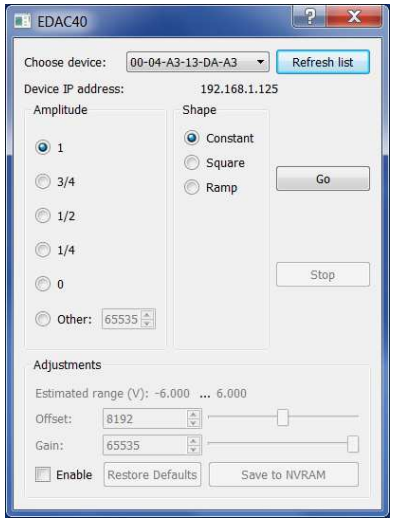

Figure 4: edac40gui – An application for test and adjustments

5 Programming of the device

5.1 Low-level programming interface

Sending packets using TCP/IP stack is a routine task and can be performed quite easyly in any reasonably modern programming language. Supporting libraries are available for most modern platforms: WinSock in Windows, BSD sockets in Unix-like operational systems and so on. Nevertheless, special library provided on accompanying CD to help the customer in crafting and sending packages to control EDAC40 devices. As the source code in C language is available, it can be compiled and linked together with user program or used as a starting point for designing more sophisticated software. Compiled version of the library in both static and dynamic (DLL) form included as well, making it easy to deploy in other programming languages, such as Pascal/Delphi. This section contains short description of functions and data structures and constants defined by the library. For complete details please refer to the source code. Sample project files for Code::Blocks IDE are included and can be easily used for recompilation and modifications.

5.1.1 Deploying the library

User source code should include library header `sdk/include/edac40.h` (and usually `winsock2.h` header as well) with preprocessor directives, the file should be located either in the same directory or in compiler search path:

```
#include <winsock2.h>
#include "edac40.h"
```

The library itself `sdk/lib/libedac40.a` should be linked with the application. In addition, in Windows environment linking with `winsock` library is required. For most compilers it is achieved with combination of `-ledac40 -lws2_32` and `-Lsdk/lib/` options.

5.1.2 Initializing the library

Before calling any other functions of EDAC40 library initialize socket mechanism with a call of `edac40_init()`. This function usually must be called only once in the beginning of the program. It has no arguments and returns always 0.

At the end of the program it is recommended, though not strictly required, to shut down the socket machinery with a call of `edac40_finish()` function.

5.1.3 Listing and finding available devices

As the module(s) are connected to a network and not directly to a computer, it has special built-in mechanism allowing one to find out all devices within the reach (that is in broadcast network domain). The method utilizes Microchip Inc. “discover” protocol. When recieves “discover” UDP packet broadcasted, EDAC40 module responses with special string, containing module type (“EDAC40”), 6-byte MAC address (in the form “XX-XX-XX-XX-XX”) and IP address (dot separated “XXX.XXX.XXX.XXX”). The MAC address is unique identifier, managed by the Institute of Electrical and Electronics Engineers (IEEE). In the case of EDAC40 modules it serves also as device serial number and marked at the bottom part of its enclosure as well as on the PCB. IP address is naturally used to send packet by means of TCP/IP stack suit.

EDAC40 library provides two functions to facilitate listing and finding devices. Function `edac40_list_devices` has the following prototype:

```
int edac40_list_devices(edac40_list_node *devices, int max_device_num,
                        int discover_timeout, int discover_attempts);
```

The function accepts an array `devices` containing `max_device_num` elements of structure type with obvious meaning:

```
typedef struct
{
   char IPAddress[16];
   char MACAddress[18];
} edac40_list_node;
```

Two additional parameters are: `discover_timeout` – period of time given in milliseconds within which devices are expected to response (typical useful values are of order of hundreds of milliseconds) and `discover_attempts` – number of times the broadcast "discover" packet is send (one attempt is usually just enough except of the case when used in unreliable network carrying pretty heavy traffic).

The function fills provided array with data and returns number of found devices or zero, if no devices found.

The second function has limited functionality but simpler to use. Its prototype is:

```
char* edac40_find_device(char *macaddress)
```

It accepts string with MAC address as a single argument and returns a pointer to the string with IP address or NULL pointer in the case if there is no device with such MAC exists. Please note, that function allocates memory for storing the string, so the user should take care of memory disposing.

The following compete example demonstrates listing of EDAC40 devices present in the network.

```
/* file edac40_list.c */
#include <stdio.h>
#include "edac40.h"
#define EDAC40_MAXN 10
#define EDAC40_DISCOVER_TIMEOUT 500 // milliseconds
#define EDAC40_DISCOVER_ATTEMPTS 1

int main()
{
   int device_num,i;
   edac40_list_node edac40_list[EDAC40_MAXN];
   edac40_init();
   printf("Detecting EDAC40 devices...\n");
   device_num=edac40_list_devices(edac40_list, EDAC40_MAXN, EDAC40_DISCOVER_TIMEOUT,
                                  EDAC40_DISCOVER_ATTEMPTS);
   printf("Detected %d EDAC40 unit(s).\n",device_num);
   for(i=0; i<device_num; i++)
      printf("Unit %d: IP Address: %s, MAC Address:%s\n",
             i,edac40_list[i].IPAddress,edac40_list[i].MACAddress);
   edac40_finish();
   return 0;
}
```

5.1.4 Connecting to a EDAC40 unit

Connection to the device is performed by the function with the following prototype:

```
SOCKET edac40_open(char *edac40_host, int use_tcp)
```

The device specified by `edac40_host` string, which is typically dot-separated presentation of units IP address, but can be symbolic host name if DNS name resolving works (configuring the DNS server is out of scope of this document). Non-zero value for second argument prescribes to use TCP protocol, otherwise UDP datagrams will be used.

The function returns socket identifier (positive integer), which is passed as parameters to other functions and should be stored in some variable for further use. In the case of failure the negative error code is returned.

Multiple connections are allowed to the same UDP socket (since actually it is "connection-less" protocol and sending of each packet treated as atomic transaction), but call to `edac40_open` using TCP protocol will fail if attempt is made to connect to the unit which is already in use.

There is no practical limitations for the number of simultaneously opened sockets.

The following function:

```
void edac40_close(SOCKET edac40_socket)
```

closes EDAC40 socket specified by its single `edac40_socket` parameter. Internally it just closes Windows/BSD socket and returns no values.

5.1.5 Specific issues of using TCP protocol

While most of the program logic remain the same in case of using either TCP or UDP protocol, there is one subtle difference. In the case of UDP sending of data block with `edac40_send_packet` always successful and the function returns immediately, even if device is already switched off or data could not be delivered for some other reason. When socket is open for TCP communication, the delivery is guaranteed and function returns when data accepted and acknowledged or (in the case of some network problem) when timeout reached. This kind of behavior is usually referred as "blocking write". Amount of time function spend in blocking call could be specified with the following function:

```
void edac40_set_timeout(SOCKET edac40_socket, long milliseconds)
```

It sets timeout period of `milliseconds` (second argument) for the socket specified by its first argument. Default value is 1000 (1 second).

5.1.6 Forming data packets

Data send to the unit in blocks of fixed format (described in section 3), there are several "helper" function which facilitate forming those blocks for some typical situations.

```
int edac40_prepare_packet(edac40_channel_value *channel_list, int channel_num,
                          char **edac40_packet)
```

The function forms data block based on a list of channel number – value pairs. The list is given as first parameter (`channel_list`) and consist of `channel_num` elements of structure type

```
typedef struct
{
    int channel;
    uint16_t value;
} edac40_channel_value;
```

channels are given as integers (`channel`), starting from zero. Those are "logical" channel numbers as labeled on rear panel (minus 1). Values (`value`) are represented by unsigned 16-bit integers. Data block address is returned in the `edac40_packet`, provided by caller. Packet size in bytes is returned as the function result. The function is most useful for setting irregular group of channels. As the function dynamically allocate memory for the data block, it is responsibility of the user to dispose it when not needed anymore.

The function with the prototype

```
int edac40_prepare_packet_from_array(unsigned value[40], int command_code,
                                     char **edac40_packet)
```

is more convenient when it is required to set output for all 40 channels. It takes an array `value[40]` of unsigned integers values and `command_code` as arguments. It can be used for settings output, gain and offset, predefined constants for codes are: `EDAC40_SET_VALUE`, `EDAC40_SET_OFFSET`, `EDAC40_SET_GAIN`, `EDAC40_SET_OFFSET_DACS` Data block address is returned in the pointer, provided by caller. Function returns the size of block in bytes, which is always 86. The user should take care of data block memory freeing after use.

```
int edac40_prepare_packet_fill(unsigned value, int command_code, char **edac40_packet)
```

– even more simple version fills the block with the same `value` for all channels. Other parameters meaning is the same as for `edac40_prepare_packet_from_array`.

5.1.7 Sending data

Data for EDAC40 are send in blocks with a function:

```
int edac40_send_packet(SOCKET edac40_socket, char *edac40_packet, int edac40_packet_size)
```

It sends packet of `edac40_packet_size` bytes, specified by its address `edac40_packet` to socket `edac40_socket`.

It is possible to send data (DAC value, gain, offset, Offset DAC value) for only one channel with the function:

```
int edac40_set(SOCKET edac40_socket, int command_code, int channel, unsigned value);
```

Socket is identified by `edac40_socket`, `command_code` describes an action and could be choosen from predefined constants `EDAC40_SET_VALUE`, `EDAC40_SET_OFFSET`, `EDAC40_SET_GAIN`, `EDAC40_SET_OFFSET_DACS`, the command applies for single `channel`, numbered from zero. Though all data in principle could be send with this function, the user is strongly discouraged to do so, as it creates much network overhead and significantly decreases throughput. Please use `edac40_prepare_packet` and `edac40_send_packet` instead.

5.1.8 Storing parameters into NVRAM

EDAC40 features very flexible adjustment of output voltage range. They could be stored in non-volatile and automatically reload when the module is powered-up again. The function

```
int edac40_save_defaults(SOCKET edac40_socket);
```

requires one argument – socket handler, it simply sends single 8-byte packet to store all current settings.

```
int  edac40_restore_defaults(SOCKET edac40_socket)
```

– Loads factory defaults and stores them into NVRAM.

5.1.9 A complete example

The example program sends packets corresponding minimum (0) and maximum (65536) values to first found EDAC40 module thus generating square wave signal. Period between packets is measured with RDTSC CPU instruction, present value of 800000 gives frequency of approximately 2kHz.

```
/* file: edac40_sqwave.c */
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include "winsock2.h"
#include <stdint.h>
#include "edac40.h"

#define EDAC40_MAXN 10
#define EDAC40_DISCOVER_TIMEOUT 500 // milliseconds
#define EDAC40_DISCOVER_ATTEMPTS 1

/* ReaD the Time Stamp Counter  */
inline unsigned long long int rdtsc(void)
{
   unsigned a, d;
   __asm__ volatile("rdtsc" : "=a" (a), "=d" (d));
   return ((unsigned long long)a) | (((unsigned long long)d) << 32);;
}

int main()
{
   int nb;
   int tmp_ch_n;
   SOCKET edac40_socket;
   edac40_list_node edac40_list[EDAC40_MAXN];
   edac40_channel_value dac_data0[40], dac_data1[40];
   char *buf0, *buf1;
   int i,buf_len,device_num;
   unsigned long long int t1,t2;
   fprintf(stderr,"Looking for EDAC40 devices...\n");
   edac40_init();
   device_num=edac40_list_devices(edac40_list, EDAC40_MAXN,
                        EDAC40_DISCOVER_TIMEOUT, EDAC40_DISCOVER_ATTEMPTS);
   if(device_num<=0)
        {
            fprintf(stderr,"No EDAC40 units detected.\n");
            exit(1);
        }
      else
        {
            fprintf(stderr,"Using first found EDAC40 unit for square wave generation"
                           "(Ctrl-C to terminate).\n");
        }
   edac40_socket=edac40_open(edac40_list[0].IPAddress,0);
   edac40_set_timeout(edac40_socket,2000); // 2s
   tmp_ch_n=40;
```

```
   for(i=0;i<tmp_ch_n;i++)
     {
         dac_data0[i].channel=dac_data1[i].channel=i;
         dac_data0[i].value=0;
         dac_data1[i].value=0xFFFF;
     }
   edac40_prepare_packet(dac_data0,tmp_ch_n,&buf0);
   buf_len=edac40_prepare_packet(dac_data1,tmp_ch_n,&buf1);
   i=0;
   t2=rdtsc();
   while(1)
       {
        t1=rdtsc();
        if(t1-t2<800000ull) continue;
        t2=t1;
        nb=edac40_send_packet(edac40_socket, ((i++)&1)?buf0:buf1, buf_len);
        if(nb<=0)
           {
              printf("\nConnection lost.\n");
              exit(1);
           }
       }
   free(buf0);
   free(buf1);
   edac40_close(edac40_socket);
   edac40_finish();
   return 0;
}
```

5.2 Higher-level examples

A number of simple demo programs are provided on the accompanying CD in the form of both source code and binary executables. They are built upon the `edac40` library and can be used to check functionality of the system (DAC unit, adaptive mirror, high-voltage amplifier) as well as a base for developing some application software. These programs include (their structure and use are pretty obvious):

am_set applies the same voltage given as argument to all actuators.

rotate sets the voltage given as argument to all channels of OKO mirror, one by one.

set_channel sets given voltage into one given channel, 0s to others.

ring1_set activates outer ring of actuators. Usable for 19-, 39- and 79-channel mirrors.

ring2_set activates second from outside ring of actuators. Usable for 19-, 39- and 79-channel mirrors.

19_set sets given value to 20–37 channels. Usable for OKO 37ch mirrors.

7_set sets given value to 8-19 channels. For OKO 19ch mirror

degauss set maximum voltage to all mirror channels periodically. Can be used for removing effect of response hysteresis of PDM mirrors.

pairs applies maximum voltages to actuator pairs of linear 20-channel OKO mirror.

random_test activates random combinations of the channels in infinite loop.

smiley19 supposed to show smiley face on the surface of OKO 19-ch mirror.

smiley37 supposed to show smiley face on the surface of OKO 37-ch mirror.

print_pinout outputs connector pin to mirror channels assignments in human readable representation.

edac40_list polls EDAC40 devices in the network using “discover” protocol and lists their IP addresses along with hardware (MAC) addresses.

edac40_sqwave generate square wave signal in all 40 channels of the EDAC40 unit.

(The last two programs are not directly related to adaptive mirrors).

In the cases of mirrors with the number of actuators greater than 40 several EDAC40 units are used. MAC address of the units should be listed (one per line, hexadecimal bytes separated by dash character, i.e. XX-XX-XX-XX-XX-XX) in the text file `sernum.ini` present in the same directory as an executable program.

The programs are written in plain ("vanilla") C with minimal use of common libraries. They can be easily compiled with any modern C compiler such as GCC, MinGW (publicly available) or Microsoft C. The programs (with some exceptions) are applicable to all adaptive mirrors developed by Flexible Optical B.V. both MMDM and PDM. Data for all of the mirror types (such as the number of actuators and logical mapping between DAC channels and actuator number) along with the simple function for DAC are declared and defined in C header file `mirror_edac40.h` and source file `mirror_edac40.c` correspondingly. Parameters appropriate for particular mirror device are selected by means of conditional compilation (`#define` – `#ifdef` mechanism). For the moment valid types are following (one should be defined during compilation): `MMDM_37CH`, `MMDM_39CH_30MM`, `MMDM_79CH_30MM`, `MMDM_79CH_40MM`, `MMDM_79CH_50MM` ((membrane (MMDM) mirrors); `PIEZO_19CH`, `PIEZO_37CH`, `PIEZO_37CH_2005`, `PIEZO_37CH_50MM`, `PIEZO_37CH_50MM_2008`, `PIEZO_79CH_50MM`, `PIEZO_109CH_50MM` (piezoelectric actuator based mirrors); `PIEZO_19LO_30`, `PIEZO_LO18CH` (PDM mirrors, optimized for low-order aberrations); `MMDM_LIN19CH`, `PIEZO_LIN20CH` (linear MMDM and PDM mirrors); `MMDM_17TT` (membrane mirror with built-in tip-tilt).

Any of the above-mentioned programs can be easily compiled from a command line, for example `rotate.c` program for 37-channel MMDM mirror can be compiled with MinGW GCC under Windows as follows:

```
gcc -DMMDM_37CH rotate.c mirror_edac40.c edac40.c -I../include -lws2_32 -o rotate.exe
```

For users more comfortable with IDEs a set of Code::Block projects and workspace file loading all the projects are provided. Those can be used to generate a complete set of executables for all mirror type (distributed among corresponding directories) by pressing a single button.