MVE

Modular Visualisation Environment

Programming documentation

version 1.01

**Centre of Computer Graphics
and Data Visualisation**

University of West Bohemia
Univerzitní 8, Box 314, 30614 Plzeň
Czech Republic

Centre of Computer Graphics and Data Visualisation

http://herakles.zcu.cz

Staff

Prof. ing. Václav Skala, CSc.
Doc. dr. ing. Ivana Kolingerová
Ing. Jiří Dobrý
Ing. Martin Franc
Ing. Jan Hrádek
Ing. Marek Krejza
Ing. Martin Kuchař
Ing. Pavel Maur
Ing. Ladislav Pešička
Ing. Michal Roušal
Ing. Tomáš Jirka
Ing. Radek Sviták
Martin Čermák
Jindřich Suja

February 25, 2000

1. Table of contents

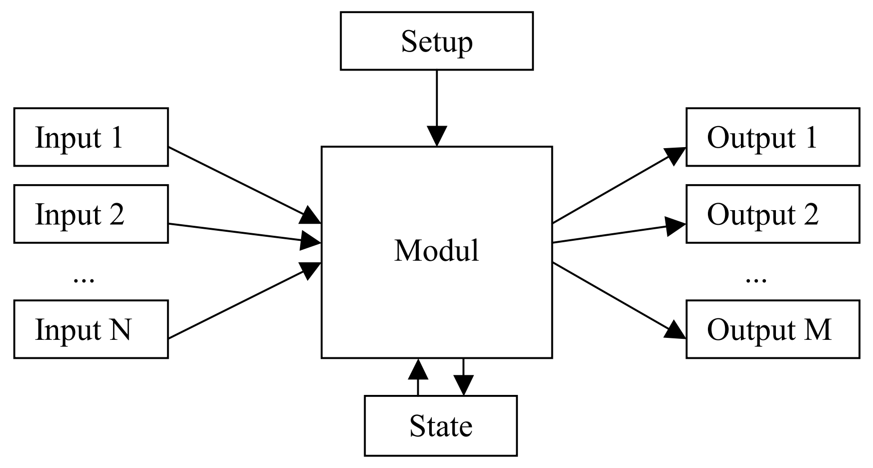
2. How to create new MVE modules

3. What is this all about

This system is provided to allow user to visually design schemes from computing/loading/rendering modules to create own "application". Those applications are also executed by this system.

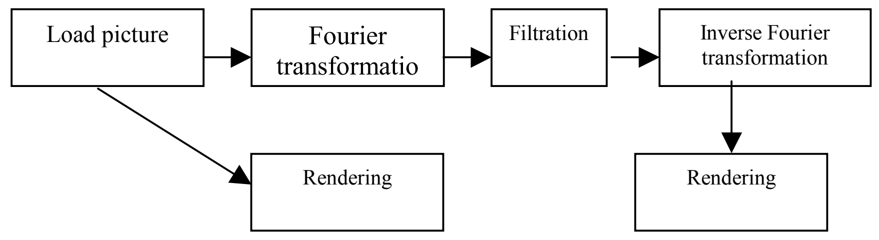
Module means "object" which can from his input data and some setup produce some output data. Some modules can have no input (loading from drive for example) and of course some modules can have no output (modules for data visualization "renderers"). On figure 1.1 we can see an example of a most common module structure.

Figure 1.1



The module connecting process means that the user interactively connects outputs of one module to inputs of another module. For such process a design system is provided (description is not included in this documentation.

Figure 1.2



After the execution of this scheme, two windows should appear on the screen, one with the original picture and one with the filtered picture.

This application can execute designed schemes of modules and transfer data between these modules. There is also implemented a mechanism for parallel execution of schemes, where is that possible.

Main aim of this application is to make some standard data formats (structures) and eliminate redundant and no important work on chosen problem (algorithm).

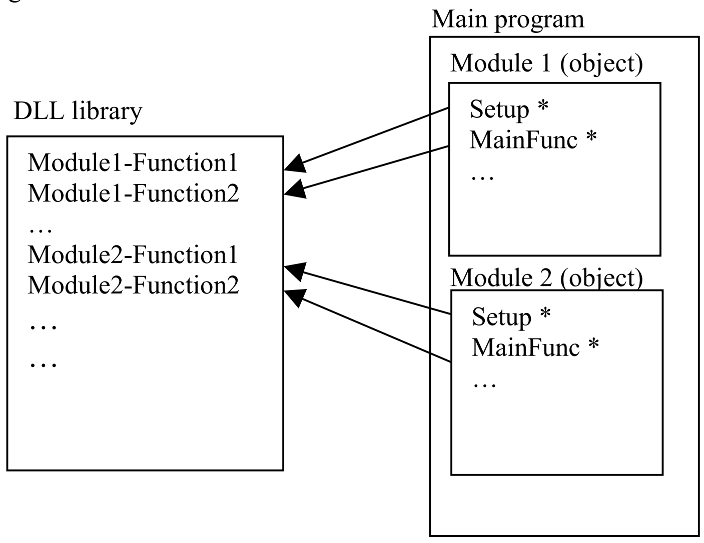
For example, in case of picture filtration (as you can see on figure 1.2), it is not necessary to implement always-new part for loading or rendering pictures. When some defined data structures and programming structure will be used, already existing modules can be added the scheme (if they are solving problem we need) and the editor can connect them together with your own module.

4. How MVE works

System is using DLL libraries under MS Windows™ environment. Modules can be created with all programming languages, which enable creating DLLs, like MS VC++, Borland Delphi or Borland C++ Builder.

Every module is represented by the set of functions of given name (module name + suffix). Those functions are included in one DLL and represents complete function ability of module. One DLL library can of course contain more modules. For internal use the modules are represented in system as objects, see figure 1.3.

Figure 1.3



This solution has one problem, which is that all memory that is allocated by functions from DLL library must be deallocated by functions from the same DLL library. This is involving necessity that you must write deallocating routines for all data structures which are allocated by your module and returned as outputs of this module.

5. Module creation

You have to create in your programming environment empty DLL project as first. All functions have their names given by name of module and by name of particular type of function. Functions for memory deallocation are necessary because of memory management mentioned above. Also is necessary to set structure member alignment to "1 BYTE" and calling convention to "stdcall" for all created DLLs.

In next part will be described all required functions for one module, their parameters and return values. Lets anticipate that we already have created empty project and now we need only to add functions.

5.1. Function used to get information about modules stored in DLL

When the empty project is created we have to add functions, which are used to get information about modules stored in this DLL (number of modules, counts of their inputs and outputs and data types). This part is mostly theoretical and real data types etc. will be in sections for given programming languages.

Pointer to information about modules Get_Modules()
This function is only one in DLL library and is used to give information about contained modules to system itself.
Return value Function return pointer to structure with information about modules in DLL (type of module, name of module, number of its inputs and outputs etc.)

Numeric values of existing data types and module types, which have to be set, are in the following table.

Data types:		Module types:	
Undefined	0	Loader	1
Points	1	ComputingModule	2
Triangles	2	Saver	3
Tetrahedras	3	Renderer	10
Image	4		
Volume	5		
Freq	6		

On our internet pages (WWW project pages) should be accessible actual versions of files *Types.def* and *Modules.def* with this numbers.

Free_DLL_Descr(pointer to info about modules)
This function is here to free information data that has been returned by function *Get_Modules().*

5.2. Module functions

Now we have a DLL project with registered modules that will be included in this DLL. You have to add functions, which realize the execution part of module or user setup of module (file name, some parameters). Each module consists from five obligatory and one non-obligatory function. Their names are created from name of module and suffixes, which are declared for function types in header files (*MVE_Include.h, or MVE_Include.pas*). There are also parameters and return values of those functions. All functions must be reentrant to enable parallel execution.

Names of functions consist from module name and suffix, for this part of documentation is name of module replaced by string *ModuleName*. Real data types in function declaration are also replaced by some description.

Pointer to new module setup ModuleName_SETUP_FUNC(*Pointer to old module setup)*
This function gives to the user the way, how to set some parameters for the module. This function can be called in the editor and should display some dialog for this setup.
Pointer to old module setup Pointer to the data structure with old module setup. This data are read-only, this pointer is set to NULL if module has no previous setup.
Return value Function returns pointer to structure with new module setup. There is also member value with the size of this setup data in the structure (see chapter 8.2). The setup data must be consistent block of data, it means that there can't be any others pointers or references. This is because the setup data are copied during the design of schemes and are saved together with schemes. When no setup has been done (dialog was ended by *Cance*l button for example) you can return NULL.

Pointer to module output data ModuleName_MAIN_MODULE_FUNC(*Pointer to module setup data,* *Pointer to module input data,* *Module state data)*
This is main function of module and it realizes whole execution part of module.
Pointer to module setup data User setup data returned by previous function *ModuleName_SETUP_FUNC*. When module has still no setup data this parameter is NULL.
Pointer to module input data Data to input of this module can come from other modules, of course only when this module needs some input data for his work. When module has some input data than this data are read-only. Only exception is *Data_State* member (see chapter 7.2).
Module state data Pointer to pointer, which can be used to point to data with stored module state. This is just replacement of global pointer in DLL which can't be used because of parallel execution. This parameter is here only for iteration modules but in current version are no iteration modules. We don't recommend to use this parameter in this version.
Return value Return value of this module is pointer to data structure containing all outputs of the module. All data types (their numbers) must be same as are specified in description data returned by function *Get_Modules()*. For data structures description see chapters 8.1 and 8.2.

ModuleName_FREE_SETUP_DATA(Pointer to module setup data)
Function for deallocation of data with module setup from *ModuleName_SETUP_FUNC.*
Pointer to module setup data Pointer to data to deallocate.
Return value 0 OK, other failed

ModuleName_FREE_DATA(Pointer to module output data)
This function must deallocate all data returned by *ModuleName_MAIN_MODULE_FUNC* as output data.
Pointer to module output data Pointer to data structure with data to free.
Return value 0 OK, other failed.

ModuleName_FREE_STATE(module state)
Function to free state data of module, if used.
Module state Pointer to pointer to module state data.
Return value 0 OK, other failed.

Pointer to a string ModuleName_HELP_FUNC()
This function returns some base info about module (some simple help for example). This function is not obligatory.
Return value “C” String with information (end line chars are 13,10 (\r\n)).

6. Create module with MS VC++ 6.0

6.1. Create project

Lets premise, that you want to create the DLL with two modules, one module for data loading (loading set of points) and one for some data processing (triangle mesh generating for example). These modules can be named „*PointLoader*“, and „*Triangulation*“

1. File – New
2. Projects – „MFC *AppWizard (DLL)“* (or „*Win32 Dynamic-Link Library*“ when you don't want use any MFC classes) – Project Name – „*TestModules*“
3. MFC Extension DLL (using shared MFC DLL) or Regular DLL using shared (statical) MFC DLL

Copy the *MVE_Include.h* file that contains all definitions and some macros to the directory with the project and add this file to the project.

Add those lines:

StdAfx.h
... *#include "MVE_Include.h"* ...

Global setting changes:

Set structure member alignment to 1 byte and calling convention to ***__**stdcall*.

1. Project – Settings
2. Setting for: *All Configurations*
3. C/C++ - Category: *Code Generation*
4. Calling convention: ***__**stdcall*
5. Struct member alignment: *1 Byte, or #pragma pack(1)*

6.2. How to get information about modules in the DLL

Small example for our DLL library:

```
TestModules.cpp
...
DLL_DESCR* __stdcall Get_Modules(void)
{
MODULE_DEF_INIT

ADD_MODULE(1," PointLoader","This module loads set of modules")
ADD_OUTPUT(1,"Loaded points")

ADD_MODULE(2," Triangulation","Triangulation on set of points")
ADD_OUTPUT(2,"Triangle mesh")
ADD_INPUT(1," Points for triangulation ")

MODULE_DEF_DONE
}

int __stdcall Free_DLL_Descr(DLL_DESCR *What)
{
FREE_DLL_INFO(What)     // or write deallocation yourself
return 0;
}
...
```

Macros *MODULE_DEF_INIT*, *ADD_MODULE*, *ADD_INPUT*, *ADD_OUTPUT*, *MODULE_DEF_DONE* and *FREE_DLL_INFO* are defined only for C programming language. For Pascal you have to allocate, set members and also deallocate all data structures. Functionality of the macros is noticeable from example above.

Parameters of macros:

ADD_MODULE(Module type, Module description)

Module type	- number representing module type
Module description	- string describing this module

ADD_OUTPUT(Data type, IO Description), ADD_INPUT(...)

Data type	- number representing data type for this input/output
IO Description	- string describing this input/output

The values for parameters "*Module type and Data type*" are given (see chapter 3.1) and by these values the checking in the editor is done.

Add these lines to *TestModules.def* file, which is used for defining exported functions.

TestModules.def
...
Get_Modules   @1
Free_DLL_Descr   @2

6.3. Main module functions

In this case it will be following functions:

For module *PointLoader*:

PointLoader_SETUP_FUNC
PointLoader_MAIN_MODULE_FUNC
PointLoader_FREE_SETUP_DATA
PointLoader_FREE_DATA
PointLoader_FREE_STATE
PointLoader_HELP_FUNC

For module *Triangulation*:

Triangulation_SETUP_FUNC
Triangulation_MAIN_MODULE_FUNC
Triangulation_FREE_SETUP_DATA
Triangulation_FREE_DATA
Triangulation_FREE_STATE
Triangulation_ HELP_FUNC

Example of functions headers for module *PointLoader*:

TestModules.cpp
CUSTOM_MODULE_DATA* __stdcall PointLoader_SETUP_FUNC(*CUSTOM_MODULE_DATA* oldSetup)*
DATA_DESCRIPTORS __stdcall PointLoader_MAIN_MODULE_FUNC(CUSTOM_MODULE_DATA* setupData, DATA_DESCRIPTORS* inData, void** stateData)*
int __stdcall PointLoader_FREE_SETUP_DATA(CUSTOM_MODULE_DATA oldSetup)*
int __stdcall PointLoader_FREE_DATA(DATA_DESCRIPTORS oldData)*
int __stdcall PointLoader_FREE_STATE(void oldState)*
char __stdcall PointLoader_HELP_FUNC(void)*

At the end add those lines for function export to *TestModules.def* file.

```
TestModules.def
...
PointLoader_SETUP_FUNC               @3
PointLoader_MAIN_MODULE_FUNC         @4
PointLoader_FREE_SETUP_DATA          @5
PointLoader_FREE_DATA                @6
PointLoader_FREE_STATE               @7
PointLoader_HELP_FUNC                @8
Triangulation_SETUP_FUNC             @9
Triangulation_MAIN_MODULE_FUNC       @10
Triangulation_FREE_SETUP_DATA        @11
Triangulation_FREE_DATA              @12
Triangulation_FREE_STATE             @13
Triangulation_ HELP_FUNC             @14
```

6.4. Usage of modules

After build of the DLL you have *TestModules.DLL* containing 2 modules. It should export all functions that were specified in *TestModules.def* file. This DLL is easy to add to the editor, where you can use your modules for designing new schemes.

7. Create module with Delphi v 4.0

Here we will describe only specific parts for Delphi.

7.1. Create project

To create empty DLL project choose:

1. File – New...
2. New - DLL

Copy the file *MVE_Include.pas*, to directory with project and add them to project.

Because this DLL needs to work with NULL terminated strings you must add also *ShareMem* to units. When you want to use DLL with modules from Delphi you must copy *DELPHIMM.DLL* with it.

TestModules.dpr
... *uses ShareMem, ... , MVE_Include, ...* ...

Project settings changes:

Set record fields align to 1 byte.

1. Project – Options...
2. Compiler – Code Generation – **uncheck** Aligned record fields

7.2. How to get information about modules in the DLL

You have to add all functions described in chapters 3.1 and 3.2 to this DLL. All those functions must be of same type as are described in *MVE_Include.pas*. It is also important to export all functions with „*Calling convention*“ **stdcall** (see file *MVE_Include.pas*).

Example of function headers for the module *PointLoader*:

TestModules.dpr

```
function Get_Modules: PDLL_DESCR; stdcall;
function Free_DLL_Descr(mDescr: PDLL_DESCR): Integer; stdcall;

function PointLoader_SETUP_FUNC (
            What : PCUSTOM_MODULE_DATA) : PCUSTOM_MODULE_DATA; stdcall;
function PointLoader_MAIN_MODULE_FUNC (
            mData : PCUSTOM_MODULE_DATA;
            mDataDescr : PDATA_DESCRIPTORS;
            mState : Pointer) : PDATA_DESCRIPTORS; stdcall;
function PointLoader_FREE_SETUP_DATA (
            What : PCUSTOM_MODULE_DATA) : Integer; stdcall;
function PointLoader_FREE_DATA (
            What : PDATA_DESCRIPTORS) : Integer; stdcall;
function PointLoader_FREE_STATE (What : pointer) : Integer; stdcall;
function PointLoader_HELP_FUNC () : PChar; stdcall;
```

To export functions out from the DLL use ***exports*** command**.**

TestModules.dpr

```
...
exports
  Get_Modules                          index 1,
  Free_DLL_Descr                       index 2,

 PointLoader_MAIN_MODULE_FUNC          index 3,
 PointLoader_SETUP_FUNC                index 4,
 PointLoader_FREE_DATA                 index 5,
 PointLoader_FREE_SETUP_DATA           index 6,
 PointLoader_FREE_STATE                index 7,
PointLoader_HELP_FUNC                  index 8,

 Triangulation_MAIN_MODULE_FUNC        index 9,
 Triangulation_SETUP_FUNC              index 10,
 Triangulation_FREE_DATA               index 11,
 Triangulation_FREE_SETUP_DATA         index 12,
 Triangulation_FREE_STATE              index 13;
Triangulation_ HELP_FUNC               index 14;

Begin
....
end.
```

When using Delphi you can't use macros in functions *Get_Modules* and *Free_DLL_Descr*, so you have to set all data structures yourself. Meaning of these functions is same as in chapters 4.2 a 4.3.

Data structures are same (binary) as was described and only difference is in the names of the data types in C and Pascal. For strings is good to use *PChar* type in Delphi.

Compatible data types Delphi vs. C (MS VC++):

Delphi	C
Shortint	Char
Smallint	Short
Longint	Long
Byte	Unsigned char
Word	Unsigned short
Integer	Int
Cardinal	Unsigned long
Single	Float
Double	Double
Pointer	Void*
Char	Char

8. Create module with Borland C++Builder 3.0

8.1. Create project

Only two steps are different in this case from VC++ section:

1. File – New...
2. New - DLL

Copy the *MVE_Include.h* file to the directory with project.

Add this line:

TestModules.cpp
... *#include "MVE_Include.h"* ...

Problems with data alignment in VCL library can be solved in this way:
Every inclusion of VCL headers must be surrounded by pragma tags.

```
#pragma option -a4  // alignment 4 bytes
 #include <vcl.h>
 #pragma option -a1  // alignment 1 byte
```

Project settings changes:

Set structures member alignment to 1 byte, and __stdcall as standard „Calling convention“ for all functions in project.

3. Project – Options
4. Advanced Compiler
5. Data alignment: *Byte*
6. Calling convention: *Standard call*

8.2. How to get information about modules in the DLL

Add to project all functions described in chapters 3.1 and 3.2. All this functions should have „*Calling convention*“ **stdcall** (see file *MVE_Include.h*).

Example of function headers for module *PointLoader*:

TestModules.cpp
*DLL_DESCR * __export __stdcall Get_Modules(void)*
*int __export __stdcall Free_DLL_Descr(DLL_DESCR *What)*
*CUSTOM_MODULE_DATA * __export __stdcall PointLoader_SETUP_FUNC(CUSTOM_MODULE_DATA *What)*
*DATA_DESCRIPTORS * __export __stdcall PointLoader_MAIN_MODULE_FUNC(CUSTOM_MODULE_DATA *CustomData, DATA_DESCRIPTORS *Descriptors, void **State)*
*int __export __stdcall PointLoader_FREE_SETUP_DATA(CUSTOM_MODULE_DATA *What)*
*int __export __stdcall PointLoader_FREE_DATA(DATA_DESCRIPTORS *Descriptors)*
*int __export __stdcall PointLoader_FREE_STATE(void *)*
*char*__export __stdcall PointLoader_HELP_FUNC(void)*

As you can see on example, export of functions is done by __export directive.

9. Description of used data structures

9.1. Data structures used to return information about modules in DLLs.

These structures are used by *Get_Modules* function to return information about the modules contained in one DLL library. When the macros for C language are used, there is no need to read following lines. But when you are using Delphi than, you have to set these structures yourself.

Structure describing one input/output of module.

C	*Pascal*

typedef struct {
    int IO_Type;
    *char *Descr;*
} IO_CONN;
	PIO_CONN = ^IO_CONN;
	IO_CONN = record
	    IO_Type : Integer;
	    Descr : PChar;
	end;

- *IO_Type* - Number of the data type for input/output
- *Descr* - Short description shown by tooltip in editor

Structure describing all inputs/outputs of module

C	*Pascal*

typedef struct {
    int IO_Num;
    *IO_CONN *IO_Types;*
} IO_DESCR;
	AIO_CONN = array[0..1] of IO_CONN;
	PAIO_CONN = ^AIO_CONN;
	
	PIO_DESCR = ^IO_DESCR;
	IO_DESCR = record
	    IO_Num : Integer;
	    IO_Types : PAIO_CONN;
	end;

- *IO_Types* - Array (pointer to first item) of the structures *IO_CONN*
- *IO_Num* - Item count of the array *IO_Types*

Structure describing one module

C	*Pascal*

typedef struct {
  int Module_Type;
  *char *Module_Name;*
  *char *Module_Descr;*
  *IO_DESCR *Inputs;*
  *IO_DESCR *Outputs;*
} MODULE_DESCR;
	PMODULE_DESCR = ^MODULE_DESCR;
	MODULE_DESCR = record
	  Module_Type : Integer;
	  Module_Name : PChar;
	  Module_Descr : PChar;
	  Inputs : PIO_DESCR;
	  Outputs : PIO_DESCR;
	end;

• *Module_Type*	- Module type (number) (see chapter 4.2).
• *Module_Name*	- Module name (Used as prefix for names of module functions)
• *Module_Descr*	- Short description of module (used as tooltip in editor)
• *Inputs*	- Pointer to the structure IO_DESCR with info about inputs
• *Outputs*	- Pointer to the structure IO_DESCR with info about outputs

Structure describing all modules in one DLL

C	*Pascal*

typedef struct {
  int Num_Descr;
  *MODULE_DESCR *Descriptors;*
} DLL_DESCR;
	AMODULE_DESCR = array[0..1] of MODULE_DESCR;
	PAMODULE_DESCR = ^AMODULE_DESCR;
	
	PDLL_DESCR = ^DLL_DESCR;
	DLL_DESCR = record
	  Num_Descr : Integer;
	  Descriptors : PAMODULE_DESCR;
	end;

• *Descriptors*	- Array (pointer to first item) of structures *MODULE_DESCR*
• *Num_Descr*	- Item count in *Descriptors* array

Same data structures are used for Delphi with only small difference in type names.

9.2. Structures for data encapsulation (for data transport between modules)

For data transfer we have to “package” all data to standard data structures.

Structure for one data input/

C	*Pascal*

typedef struct {
 int Data_Type;
 *void *Data;*
 unsigned long Data_Length;
 *void *Header;*
 unsigned long Header_Length;
 int Data_State;
} DATA_DESCR;
	PDATA_DESCR = ^DATA_DESCR;
	DATA_DESCR = record
	 Data_Type : Integer;
	 Data : Pointer;
	 Data_Length : LongInt;
	 Header : Pointer;
	 Header_Length : LongInt;
	 Data_State : Integer;
	end;

- *Data_Type* - Type of contained data (see chapter 3.1) – required
- *Data* - Pointer to data itself – required
- *Data_Length* - Data size (in bytes) – only internal information – not required
- *Header* - Pointer to data header (doesn’t need to be used when all included in data, than set to NULL) – not required
- *Header_Length* - Header data size (in bytes) – only internal information – not required
- *Data_State* - Current state of the data (Before the module execution is this value set to 1 for all inputs/outputs, when module doesn’t need input data anymore he (his programmer) must set this value to 0 for all data inputs) – required

Data states:

0 – Data are not used (are ready to deallocate)
1 – Data are still in use (can’t be deallocate)
2 – Data can be updated (only for renderer, not use in other case)
3 – Just updating data (only for renderer, not use in other case)

Structure describing all input/output data for module

C	*Pascal*

typedef struct {
int Num_Descr;
*DATA_DESCR *Descriptors;*
} DATA_DESCRIPTORS;
	ADATA_DESCR = array[0..1] of DATA_DESCR;
	PADATA_DESCR = ^ADATA_DESCR;
	
	PDATA_DESCRIPTORS = ^DATA_DESCRIPTORS;
	DATA_DESCRIPTORS = record
	Num_Descr : Integer;
	Descriptors : PADATA_DESCR;
	end;

- *Descriptors* - Array of structures *DATA_DESCR* (representing all input/output data for module)
- *Num_Descr* - Item count in *Descriptors* array

Structure for user setup of module

C	*Pascal*

typedef struct {
*void *Data;*
unsigned long Data_Length;
} CUSTOM_MODULE_DATA;
	PCUSTOM_MODULE_DATA = ^CUSTOM_MODULE_DATA;
	CUSTOM_MODULE_DATA = record
	Data : Pointer;
	Data_Length : LongInt;
	end;

- *Data* - Pointer to the data structure with user-setup of the module. This data structure must be consistent block of data (structure on which points *Data* pointer can't contain any other pointers, if string than only static ones) – required
- *Data_Length* - Exact size of data block (structure) on which is pointing *Data* pointer – required

Remarks and suggestions send to rousal@kiv.zcu.cz