Becker & Hickl GmbH
Nahmitzer Damm
12277 Berlin
Tel. +49 30 212 800 20
Fax. +49 30 212 800 213
email: info@becker-hickl.com
http://www.becker-hickl.com

GVD

Dynamic Link Libraries

User Manual

Version 1.4, May 2014

Introduction

The **GVD** Dynamic Link Library contains all functions to control the GVD modules. The functions work under 32 or 64 bit Windows XP/Vista/7/8. Both 32 and 64-bit DLL versions are available. The program which calls the DLLs must be compiled with the compiler option 'Structure Alignment' set to '1 Byte'.

The distribution package contain the following files:

GVD32.DLL	32-bit dynamic link library main file for use on 32-bit systems
GVD32x64.DLL	32-bit dynamic link library main file for use on 64-bit systems
GVD32.LIB	import library file for Microsoft Visual C/C++ for use on 32-bit systems
GVD32x64.LIB	import library file for Microsoft Visual C/C++ for use on 64-bit systems
GVD64.DLL	64-bit dynamic link library main file for use on 64-bit systems
GVD64.LIB	import library file for Microsoft Visual C/C++ for use on 64-bit systems
GVD_DEF.H	Include file containing Types definitions, Functions Prototypes and Pre-processor statements
GVD120.INI	GVD DLL initialisation file
GVD_DLL.DOC	This description file
USE_GVD.C	The simple example of using GVD DLL functions. Choose the correct import library file to link in your compiler.

The installation program installs correct versions of the DLL depending on the operating system (32 or 64 bit systems require different DLL versions) together with the driver required to control GVD resources.

GVD-DLL Functions list

The following functions are implemented in the GVD-DLL:

Initialisation functions:

GVD_init
GVD_test_if_active
GVD_get_init_status
GVD_get_mode
GVD_set_mode
GVD_get_module_info
GVD_get_ error_string
GVD_get_version

Setup functions:

GVD_get_parameter
GVD_set_parameter
GVD_get_parameters

GVD_set_parameters
GVD_get_adjust_parameters
GVD_set_adjust_parameters
GVD_get_eeprom_data
GVD_write_eeprom_data
GVD_get_eeprom_defines
GVD_write_eeprom_defines

Control functions:

GVD_test_state
GVD_start_scan
GVD_stop_scan
GVD_prepare_scan_curve
GVD_park_beam
GVD_get_scan_info

DCS-BOX specific functions:

DCS_init
DCS_get_version
DCS_get_init_status
DCS_get_parameters
DCS_set_parameters
DCS_read_status
DCS_get_eeprom_data
DCS_write_eeprom_data

Functions listed above must be called with C calling convention which is default for C and C++ programs.
Identical set of functions is available for environments like Visual Basic which requires _stdcall calling convention. Names of these functions have 'std' letters after 'GVD', for example, GVDstd_start_scan it is _stdcall version of GVD_start_scan.
Description and behaviour of these functions are identical to the functions from the first (default) set – the only difference is calling convention.

Another set of the functions is added for use **only in LabView environment**. The names of these functions have '_LV' letters after 'GVD', for example, GVD_LV_get_eeprom_data it is LabView version of GVD_get_eeprom_data.
These functions are added to treat correctly LabView clusters with strings (inside) used as an input parameter to GVD functions – it affects all functions which have a structure with string field(s) as a parameter.
Description and behaviour of these functions are identical to the functions from the first (default) set.

Application Guide

Initialisation of the GVD Parameters

The GVD DLL Functions can control up to four GVD-120 modules on PCI bus.

Before a measurement is started the measurement parameter values must be written into the internal structures of the DLL functions (not directly visible from the user program) and sent to the control registers of the GVD module. This is accomplished by the function **GVD_init**.

The **GVD_init** function

- reads the parameters values from a specified initialisation file or set them to default if ini_file is not used
- sends the parameters values to the GVD control registers of active GVD modules
- performs a hardware test (registers test & EEPROM checksum test) of active GVD modules
- initialise a DCS-BOX if it is connected to the active GVD module

The initialisation file is an ASCII file with a structure shown in the table below. We recommend either to use the file GVD120.INI or to start with GVD120.INI and to introduce the desired changes. Module(s) can be initialised also without .ini file - in this case all parameters are set to default values and can be changed later using GVD_set_parameter(s) functions.

```
;   GVD120 settings file
;   GVD parameters have to be included in .ini file only when parameter
;   value is different from default.
;   module section (gvd_module1-4) is required for each existing GVD module

[gvd_base]
simulation = 0                  ; 0 - hardware mode(default) ,
                                ; >0 - simulation mode (see gvd_def.h for possible values)

[gvd_module1]                   ; GVD module hardware parameters

active = 1          ; module active - can be used  (default = 0 - not active)
frame_size = 9      ;
                    ; bits 0-3  - lg of frame size ( quadratic frame)
                    ;       or  lg of frame size X ( rectangular frame)
                    ;   values in range: 4 (16x16) ... 12 (4096x4096),
                    ;         ( default 9, 512x512)
                    ; bits 8-11 - lg of frame size Y ( rectangular frame)
                    ;   values in range: 4 (16) ... 14 (16384),
                    ;         ( default 9, 512)
                    ; bit 15 = 0 - quadratic frame, = 1 - rectangular frame
lasers_active = 0   ; bit 0(8) - laser 0(1) active, default 0
                   ; bit 1    - laser 0 & 1 off during flyback , default 0
                   ; bit 2    -  = 0 - laser 0 is the first laser when multiplexing lasers
                   ;           = 1 - laser 1 is the first laser when multiplexing lasers ,
                   ;           default 0, bit 2 valid for FPGA version > B2
multiplex = 0       ; lasers multiplexing mode ( bits 0-1): 0 - off,
                   ;     1 - within each pixel, 2 - after each line,
                   ;     3 - after each frame , default 0
                   ;     bits 2-15 - length of Laser1 phase in 1/100th of percent
                   ;                 when mode 1, default 5000 ( 50%)
                   ;         ( first laser = laser 0 ( FPGA version < B3 )
                   ;                       = bit 2 of laser_active  ( FPGA version > B2 )
limit_scan = 0      ; 0 - unlimited scan ( default ), 1 - scan specified no of frames
                   ; 2 - repeated scan of specified no of frames ( used with trigger)
frame_counter = 1                   ; no of frames to scan when limit_scan > 0, default = 1
scan_polarity = 0   ; default 0 ( all active low )
                   ;   bit 0 - pixel clock polarity, bit 1 - polarity of HSYNC (Line),
```

```
                    ;   bit 2 - polarity of VSYNC (Frame)
                    ;   bit = 0 - falling edge(active low)
                    ;   bit = 1 - rising  edge(active high)
scan_type = 0       ; bit 0  = 0 scan frame ( default ), = 1 - scan one line only
                    ; bit 3-1  line scanning type
                    ;       = 0 ramp continuous ( default ) ( scans also during the pixel time ),
                    ;       = 1 ramp steps (same position during the pixel) - increases min line time
                    ;       = 2 sinus continuous,
line_time = 1       ; line time in sec
zoom_factor = 1     ; zoom factor 1( default ) .. 1024,
                    ;    max scan range ( amplitude)  is divided by the factor
                    ;    max depends on frame size: 4 for frames 2048x2048,
                    ;         8 for frames 1024x1024, 16 for frames 512x512
                    ;         32 for frames smaller than 512x512
offset_x = 0        ; offset X of of the centre of zoomed area from the centre X
                    ;     of the total scan range, -100 % .. 100 % ( default 0 )
offset_y = 0        ; offset Y of of the centre of zoomed area from the centre Y
                    ;     of the total scan range, -100 % .. 100 % ( default 0 )
park_offs_x = 0   ; offset X of of the park position from the centre X
                    ;     of the total scan range, -100 % .. 100 % ( default 0 )
park_offs_y = 0   ; offset Y of of the park position from the centre Y
                    ;     of the total scan range, -100 % .. 100 % ( default 0 )
l1_power = 50     ; Laser 1 power control 0 - 100%, default 50
l2_power = 50     ; Laser 2 power control 0 - 100%, default 50
rect_zoom_x = 1     ; X zoom factor for rectangular frame, 1( default ) .. 1024,
rect_zoom_y = 1     ; Y zoom factor for rectangular frame, 1( default ) .. 1024,
scan_rate = 0       ; 0 - auto,scan rate fastest possible ( default ),
                    ; 1 - scan rate defined by line_time
park_center = 0   ; bit 0 = 1 when parked, set beam position in the centre
                    ;           of scanned area, default 0
                    ; bit 4  ( read only ) shows current state of beam parking
                    ;           1 - beam is parked, 0 - beam is not parked
scan_trigger = 0 ; External trigger condition ( FPGA version > B0 )
                    ;         bits 1 & 0 mean :
                    ;           00 - ( value 0 ) external trigger disabled (default),
                    ;           01 - ( value 1 ) external trigger active low,
                    ;           10 - ( value 2 ) external trigger active high
dcs_ctrl = 0          ; DCS box control word ( FPGA version >= A7 )
                    ;    bits 0..7   low byte of control parameters
                    ;    bits 0..2 = SPC-A total no. of routing bits
                    ;    bit  3    = SPC-A laser routing enabled
                    ;    bits 4..6 = SPC-B total no. of routing bits
                    ;    bit  7    = SPC-B laser routing enabled
                    ;    bits 8..15   high byte of control parameters
                    ;    bit  8    = Red LED state if bit 9 = 1  - 0 = off, 1 = on
                    ;    bit  9    = 0 - Red LED of Scanner shows filters switch state
                    ;              = 1 - bit 8 controls red LED state
                    ;    bit  10   = SPC-A MARK 3 enable
                    ;    bit  11   = SPC-B MARK 3 enable
                    ;    bit  12   = SPC-A DCS-Switch signals OVLD to DCC
                    ;    bit  13   = SPC-B DCS-Switch signals OVLD to DCC
                    ;    bit  15   = always 1 when DCS is controlled by software


[gvd_module2]

active = 0          ;module not active - cannot be used
```

```
[gvd_module3]

active = 0        ;module not active - cannot be used


[gvd_module4]

active = 0        ;module not active - cannot be used
```

The module will be initialised, but only when the DLL was registered correctly during installation (using License number) and when it is not in use (locked) by other application.

If, for some reasons, the module which was locked must be initialised, it can be done using the function GVD_set_mode with the parameter 'force_use' = 1.

After successful initialisation the module is locked to prevent that other application can access it.

After an **GVD_init** call we recommend to call the **GVD_test_if_active** function to check whether required GVD module is active. Only active modules can be operated further. It is recommended (but not required) to check also the initialisation status (by **GVD_get_init_status**) of the used module. In case of a wrong initialisation the initialisation status shows the reason of the error (see gvd_def.h for possible values).

Additional information about GVD modules can be obtained by calling **GVD_get_module_info** function. The function fills GVDModInfo structure which is described below.

short	module_type	module type : 120 = GVD-120
short	bus_number	PCI bus number
short	slot_number	slot number on PCI bus
short	in_use	-1 used and locked by other application, 0 - not used, 1 - in use
short	init	set to initialisation result code
unsigned short	* base_adr	lower 16 bits of base I/O address
char	serial_no[16]	module serial number

After calling the **GVD_init** function the measurement parameters from the initialisation file are present in the module control registers and in the internal data structures of the DLLs. To give the user access to the parameters, the function **GVD_get_parameters** is provided. This function transfers the parameter values from the internal structures of the DLLs into a structure of the type GVDdata (see gvd_def.h) which has to be declared by the user. The parameter values in this structure are described below.

short active	most of the library functions are executed only when module is active (not 0), default 0
short frame_size	bits 0-3 - lg of frame size (quadratic frame) or lg of frame size X (rectangular frame) values in range: 4 (16x16) ... 12 (4096x4096), (default 9, 512x512) bits 8-11 - lg of frame size Y (rectangular frame) values in range: 4 (16) ... 14 (16384),

	(default 9, 512)
	bit 15 = 0 - quadratic frame, = 1 - rectangular frame
short lasers_active	bit 0(8) - laser 0(1) active, default 0
	bit 1 - laser 0 & 1 off during flyback , default 0
	bit 2 - = 0 - laser 0 is the first laser when multiplexing lasers
	= 1 - laser 1 is the first laser when multiplexing lasers ,
	default 0, bit 2 valid for FPGA version > B2
unsigned short multiplex	lasers multiplexing mode (bits 0-1): 0 - off,
	1 - within each pixel, 2 - after each line,
	3 - after each frame , default 0
	bits 2-15 - length of Laser1 phase in 1/100th of percent
	when mode = 1, default 5000 (50%)
	(first laser = laser 0 (FPGA version < B3)
	= bit 2 of laser_active (FPGA version > B2)
short limit_scan	0 - unlimited scan (default), 1 - scan specified no of frames
	2 - repeated scan of specified no of frames (used with trigger)
unsigned short frame_counter	no of frames to scan when limit_scan > 0, default = 1
unsigned short scan_polarity	default 0 (all active low)
	bit 0 - pixel clock polarity, bit 1 - polarity of HSYNC (Line),
	bit 2 - polarity of VSYNC (Frame)
	bit = 0 - falling edge(active low)
	bit = 1 - rising edge(active high)
short scan_type	bit 0 = 0 scan frame (default), = 1 - scan one line only
	bit 3-1 line scanning type
	= 0 ramp continuous (default) (scans also during the pixel time),
	= 1 ramp steps (same position during the pixel) - increases min line time
	= 2 sinus continuous,
float line_time	line time in sec
float zoom_factor	zoom factor 1(default) .. 1024,
	max scan range (amplitude) is divided by the factor
	max depends on frame size: 4 for frames 2048x2048,
	8 for frames 1024x1024, 16 for frames 512x512
	32 for frames smaller than 512x512
float offset_x	offset X of the centre of zoomed area from the centre X
	of the total scan range, -100 % .. 100 % (default 0)
float offset_y	offset Y of the centre of zoomed area from the centre Y
	of the total scan range, -100 % .. 100 % (default 0)
float park_offs_x	offset X of the park position from the centre X
	of the total scan range, -100 % .. 100 % (default 0)
float park_offs_y	offset Y of the park position from the centre Y
	of the total scan range, -100 % .. 100 % (default 0)
float l1_power	Laser 1 power control 0 - 100%, default 50
float l2_power	Laser 2 power control 0 - 100%, default 50
float rect_zoom_x	X zoom factor for rectangular frame, 1(default) .. 1024,
float rect_zoom_y	Y zoom factor for rectangular frame, 1(default) .. 1024,
short scan_rate	0 - auto,scan rate fastest possible (default),
	1 - scan rate defined by line_time
short park_center	bit 0 = 1 when parked, set beam position in the centre
	of scanned area, default 0
	bit 4 (read only) shows current state of beam parking
	1 - beam is parked, 0 - beam is not parked
short scan_trigger	External trigger condition (FPGA version > B0)
	bits 1 & 0 mean :
	00 - (value 0) external trigger disabled (default),
	01 - (value 1) external trigger active low,
	10 - (value 2) external trigger active high

To send the complete parameters set back to the DLLs and to the GVD module (e.g. after changing parameter values) the function **GVD_set_parameters** is used. This function checks and - if required - recalculates all parameter values due to cross dependencies and hardware restrictions. Therefore, it is recommended to read the parameter values after calling **GVD_set_parameters** by **GVD_get_parameters**.

Single parameter values can be transferred to or from the DLL and module level by the functions **GVD_set_parameter** and **GVD_get_parameter**. To identify the desired parameter, the parameter identification par_id is used. The parameter identification keywords are defined in gvd_def.h.

GVD Scan Control Functions

Before scanning scan curve must be calculated and stored in GVD internal memory. This is done by **GVD_prepare_scan_curve** function. Changing most of GVD parameters causes the scan curve need to be recalculated. Therefore the function should be called after setting parameters directly before scan start.

GVD_get_scan_info function delivers information about scan curve.

GVD_start_scan starts sending the sequence of scan control signals to the GVD module outputs. Depending on the parameter 'LIMIT_SCAN' the sequence runs continuously or it runs number of frames defined by 'FRAME_COUNTER'.

GVD-120 module checks trigger condition and if the trigger is enabled it will wait for trigger pulse before the start of scan sequence. (feature available on modules with FPGA v. B1 and newer).

To test the state of a scan sequence the **GVD_test_state** function is used. The status bits delivered by the function are listed below (see also gvd_def.h).

GVD_RUNNING	1	sequence is started
GVD_EOF_SEQ	2	sequence was finished
GVD_PARKED	0x10	beam is parked

For GVD-120 modules with FPGA version > B0 only

STATUS_WAIT_TRG	4	sequence waits for trigger
STATUS_SCAN	8	GVD delivers scan signals (when = 1)

A running scan sequence can be stopped by the **GVD_stop_scan** function.

Error Handling

Each GVD DLL function returns an error status. Return values >= 0 indicate error free execution. A value < 0 indicates that an error has occurred. The meaning of a particular error code can be found in gvd_def.h file and can be read using **GVD_get_error_string**. We recommend checking the return value after each function call.

Using DLL functions in LabView environment

Each DLL function can be called in LabView program by using ‘Call Library’ function node. If you select Configure from the shortcut menu of the node, you see a Call Library Function dialog box from which you can specify the library name or path, function name, calling conventions, parameters, and return value for the node.

You should pay special attention to choosing correct parameter types using following conversion rules:

Type in C programs	Type in LabView
char	signed 8-bit integer, byte (I8)
unsigned char	unsigned 8-bit integer, unsigned byte (U8)
short	signed 16-bit integer, word (I16)
unsigned short	unsigned 16-bit integer, unsigned word (U16)
long, int	signed 32-bit integer, long (I32)
unsigned long, int	unsigned 32-bit integer, unsigned long (U32)
__int64	signed 64-bit integer, quad (I64)
unsigned __int64	unsigned 64-bit integer, unsigned quad (U64)
float	4-byte single, single precision (SGL)
double	8-byte double, double precision (DBL)
char *	C string pointer
float *	pass Pointer to Value (Numeric, 4-byte single)

For structures defined in include file xxx_def.h user should build in LabView a proper cluster. The cluster must contain the same fields in the same order as the C structure.

If a pointer to a structure is a function parameter, you connect to the node the proper cluster and define parameter type as ‘Adapt to Type’ (with data format = ‘Handles by Value’).

Connecting clusters with the contents which do not exactly correspond to the C structure fields can cause the program crash.

Problems appear if the **structure and the corresponding cluster contain string fields -** due to the fact that LabView sends to the DLL handles to LabView string instead of the C string pointers for strings inside the cluster.

In such case special version of the DLL function must be used which is prepared especially for use in LabView. Such functions have ‘_LV’ letters after ‘XXX’ (for example XXX_LV_get_module_info), and if found in xxx_def.h file they should be used in ‘Call Library’ function node instead of the standard function.

Another solution is to write extra C code to transform these data types, create .lsb file and use it in 'Code Interface' node (CIN) instead of 'Call Library'.

Experienced LabView and C users can prepare such CINs for every external code.

Description of the GVD DLL Functions

---

short **GVD_init** (char * ini_file);

Input parameters:

* ini_file: pointer to a string containing the name of the initialisation file in use (including file name and extension) or NULL

Return value:

0 no errors, <0 error code

Description:

Before a measurement is started the measurement parameter values must be written into the internal structures of the DLL functions (not directly visible from the user program) and sent to the control registers of the GVD module. This is accomplished by the function **GVD_init**. The **GVD_init** function

- reads the parameters values from a specified initialisation file or set them to default if ini_file is NULL (not used)
- sends the parameters values to the GVD control registers of active GVD modules
- performs a hardware test (registers test & EEPROM checksum test) of active GVD modules
- initialise a DCS-BOX if it is connected to the active GVD module

The initialisation file is an ASCII file with a structure shown in the table below. We recommend either to use the file GVD120.INI or to start with GVD120.INI and to introduce the desired changes.

```
;   GVD120 settings file
;   GVD parameters have to be included in .ini file only when parameter
;   value is different from default.
;   module section (gvd_module1-4) is required for each existing GVD module

[gvd_base]
simulation = 0                  ; 0 - hardware mode(default) ,
                                ; >0 - simulation mode (see gvd_def.h for possible values)

[gvd_module1]                   ; GVD module hardware parameters

active = 1          ; module active - can be used  (default = 0 - not active)
frame_size = 9      ;
                    ; bits 0-3  - lg of frame size ( quadratic frame)
                    ;       or  lg of frame size X ( rectangular frame)
                    ;   values in range: 4 (16x16) ... 12 (4096x4096),
                    ;           ( default 9, 512x512)
                    ; bits 8-11 - lg of frame size Y ( rectangular frame)
                    ;   values in range: 4 (16) ... 14 (16384),
                    ;           ( default 9, 512)
                    ; bit 15 = 0 - quadratic frame, = 1 - rectangular frame
```

```
lasers_active = 0   ; bit 0(8) - laser 0(1) active, default 0
                    ; bit 1    - laser 0 & 1 off during flyback , default 0
                    ; bit 2    -  = 0 - laser 0 is the first laser when multiplexing lasers
                    ;             = 1 - laser 1 is the first laser when multiplexing lasers ,
                    ;              default 0, bit 2 valid for FPGA version > B2
multiplex = 0       ; lasers multiplexing mode ( bits 0-1): 0 - off,
                    ;      1 - within each pixel, 2 - after each line,
                    ;      3 - after each frame , default 0
                    ;      bits 2-15 - length of Laser1 phase in 1/100th of percent
                    ;                  when mode 1, default 5000 ( 50%)
                    ;          ( first laser = laser 0 ( FPGA version < B3 )
                    ;                          = bit 2 of laser_active  ( FPGA version > B2 )
limit_scan = 0      ; 0 - unlimited scan ( default ), 1 - scan specified no of frames
                    ; 2 - repeated scan of specified no of frames ( used with trigger)
frame_counter = 1                   ; no of frames to scan when limit_scan > 0, default = 1
scan_polarity = 0   ; default 0 ( all active low )
                    ;   bit 0 - pixel clock polarity, bit 1 - polarity of HSYNC (Line),
                    ;   bit 2 - polarity of VSYNC (Frame)
                    ;   bit = 0 - falling edge(active low)
                    ;   bit = 1 - rising  edge(active high)
scan_type = 0       ; bit 0  = 0 scan frame ( default ), = 1 - scan one line only
                    ; bit 3-1  line scanning type
                    ;      = 0 ramp continuous ( default ) ( scans also during the pixel time ),
                    ;      = 1 ramp steps (same position during the pixel) - increases min line time
                    ;      = 2 sinus continuous,
line_time = 1       ; line time in sec
zoom_factor = 1     ; zoom factor 1( default ) .. 1024,
                    ;    max scan range ( amplitude)  is divided by the factor
                    ;    max depends on frame size: 4 for frames 2048x2048,
                    ;        8 for frames 1024x1024, 16 for frames 512x512
                    ;        32 for frames smaller than 512x512
offset_x = 0        ; offset X of of the centre of zoomed area from the centre X
                    ;    of the total scan range, -100 % .. 100 % ( default 0 )
offset_y = 0        ; offset Y of of the centre of zoomed area from the centre Y
                    ;    of the total scan range, -100 % .. 100 % ( default 0 )
park_offs_x = 0   ; offset X of of the park position from the centre X
                    ;    of the total scan range, -100 % .. 100 % ( default 0 )
park_offs_y = 0   ; offset Y of of the park position from the centre Y
                    ;    of the total scan range, -100 % .. 100 % ( default 0 )
l1_power = 50     ; Laser 1 power control 0 - 100%, default 50
l2_power = 50     ; Laser 2 power control 0 - 100%, default 50
rect_zoom_x = 1     ; X zoom factor for rectangular frame, 1( default ) .. 1024,
rect_zoom_y = 1     ; Y zoom factor for rectangular frame, 1( default ) .. 1024,
scan_rate = 0       ; 0 - auto,scan rate fastest possible ( default ),
                    ; 1 - scan rate defined by line_time
park_center = 0   ; bit 0 = 1 when parked, set beam position in the centre
                    ;          of scanned area, default 0
                    ; bit 4  ( read only ) shows current state of beam parking
                    ;          1 - beam is parked, 0 - beam is not parked
scan_trigger = 0 ; External trigger condition ( FPGA version > B0 )
                    ;         bits 1 & 0 mean :
                    ;          00 - ( value 0 ) external trigger disabled (default),
                    ;          01 - ( value 1 ) external trigger active low,
                    ;          10 - ( value 2 ) external trigger active high
dcs_ctrl = 0         ; DCS box control word ( FPGA version >= A7 )
                    ;    bits 0..7   low byte of control parameters
                    ;    bits 0..2 = SPC-A total no. of routing bits
                    ;    bit  3    = SPC-A laser routing enabled
                    ;    bits 4..6 = SPC-B total no. of routing bits
```

```
;   bit  7    = SPC-B laser routing enabled
;   bits 8..15   high byte of control parameters
;   bit  8    = Red LED state if bit 9 = 1  - 0 = off, 1 = on
;   bit  9    = 0 - Red LED of Scanner shows filters switch state
;             = 1 - bit 8 controls red LED state
;   bit  10   = SPC-A MARK 3 enable
;   bit  11   = SPC-B MARK 3 enable
;   bit  12   = SPC-A DCS-Switch signals OVLD to DCC
;   bit  13   = SPC-B DCS-Switch signals OVLD to DCC
;   bit  15   = always 1 when DCS is controlled by software


[gvd_module2]

active = 0          ;module not active - cannot be used


[gvd_module3]

active = 0          ;module not active - cannot be used


[gvd_module4]

active = 0          ;module not active - cannot be used
```

The module will be initialised, but only when the DLL was registered correctly during installation (using License number) and when it is not in use (locked) by other application.

If, for some reasons, the module which was locked must be initialised, it can be done using the function GVD_set_mode with the parameter 'force_use' = 1.

After successful initialisation the module is locked to prevent that other application can access it.

After a **GVD_init** call we recommend to call the **GVD_test_if_active** function to check whether required GVD module is active. Only active module can be operated further. It is recommended (but not required) to check also the initialisation status (by **GVD_get_init_status**) of the used module. In case of a wrong initialisation the initialisation status shows the reason of the error (see gvd_def.h for possible values).

Additional information about GVD modules can be obtained by calling **GVD_get_module_info** function. The function fills GVDModInfo structure (see gvd_def.h for definition).

---

short **GVD_test_if_active** (void);

Input parameters:

mod_no module number (0 - 3)

Return value:

0 - module not active (cannot be used) , 1 - module active

Description:

The procedure returns information whether the GVD module 'mod_no' is active or not. As a result of a wrong initialisation (GVD_init function) a module can be deactivated. To find out the reason of deactivating the module, run the GVD_get_init_status function.

---

short **GVD_get_init_status**(short mod_no, short * ini_status);

Input parameters:

mod_no	module number (0 - 3)
*ini_status	pointer to the initialisation status

Return value: 0 no errors, <0 error code (see gvd_def.h)

Description:

The procedure loads the ini_status variable with the initialisation result code set by the function GVD_init for module ‘mod_no’. The possible values are shown below (see also gvd_def.h):

INIT_GVD_OK	0	initialization OK
INIT_GVD_NOT_DONE	-1	active = 0 - initialization not done
INIT_GVD_WRONG_EEP_CHKSUM	-2	wrong EEPROM checksum
INIT_GVD_HARD_TEST_ERR	-4	hardware test failed
INIT_GVD_CANT_OPEN_PCI_CARD	-5	cannot open PCI card
INIT_GVD_MOD_IN_USE	-6	module already in use
INIT_GVD_WRONG_LICENSE	-7	corrupted license key
INIT_GVD_NO_LICENSE	-8	license key not read from registry
INIT_GVD_LICENSE_NOT_VALID	-9	license is not valid for GVD DLL
INIT_GVD_LICENSE_DATE_EXP	-10	license date expired

---

short **GVD_get_mode**(void);

Input parameters:

none

Return value: current mode of DLL operation

Description:

The procedure returns current mode of DLL operation (hardware or simulation). Possible ‘mode’ values are defined in the gvd_def.h file:

#define GVD_HARD	0	hardware mode

#define GVD_SIMUL120 120 simulation mode of GVD-120

---

short **GVD_set_mode** (short mode, short force_use, int *in_use);

Input parameters:

mode: mode of DLL operation

force_use force using the module if they are locked (in use)

*in_use pointer to the table with information which module must be used

Return value: 0 no errors, <0 error code (see gvd_def.h)

Description:

The procedure is used to change the mode of the DLL operation between the hardware mode and the simulation mode. It is a low level procedure and not intended to normal use. It is used to switch the DLL to the simulation mode if hardware errors occur during the initialisation.

Table ‘in_use’ should contain entries for all 4 modules:

0 – means that the module will be unlocked and not used longer

1 – means that the module will be initialised and locked

When the Hardware Mode is requested for each of 4 possible modules:

-if ‘in_use’ entry = 1 : the proper module is locked and initialised (if it wasn’t) with the initial parameters set (from ini_file) but only when it was not locked by another application or when ‘force_use’ = 1.

-if ‘in_use’ entry = 0 : the proper module is unlocked and can be used further.

When one of the simulation modes is requested for each of 4 possible modules:

-if ‘in_use’ entry = 1 : the proper module is initialised (if it wasn’t) with the initial parameters set (from ini_file).

-if ‘in_use’ entry = 0 : the proper module is unlocked and can be used further.

Errors during the module initialisation can cause that the module is excluded from use.

Use the function GVD_get_init_status and/or GVD_get_module_info to check which modules are correctly initialised and can be use further.

Use the function GVD_get_mode to check which mode is actually set. Possible ‘mode’ values are defined in the gvd_def.h file.

---

short **GVD_get_module_info** (short mod_no , GVDModInfo * mod_info);

Input parameters:

mod_no	module number (0 - 3)
* mod_info	pointer to the result structure

Return value: 0 no errors, <0 error code (see gvd_def.h)

Description:

After calling the GVD_init function (see above) the GVDModInfo internal structures for all 4 modules are filled. This function transfers the contents of the internal structure of the DLL into a structure of the type GVDModInfo (see gvd_def.h) which has to be defined by the user. The parameters included in this structure are described below.

short	module_type	module type : 120 = GVD-120
short	bus_number	PCI bus number
short	slot_number	slot number on PCI bus
short	in_use	-1 used and locked by other application, 0 - not used, 1 - in use
short	init	set to initialisation result code
unsigned short	* base_adr	lower 16 bits of base I/O address
char	serial_no[16]	module serial number

---

short **GVD_get_error_string**(short error_id, char * dest_string, short max_length);

Input parameters:

error_id	GVD DLL error id (0 – number of GVD errors-1) (see gvd_def.h file)
*dest_string	pointer to destination string
max_length	max number of characters which can be copied to ‘dest_string’

Return value: 0: no errors, < 0: error code

The procedure copies to ‘dest_string’ the string which contains the explanation of the GVD DLL error with id equal ‘error_id’. Up to ‘max_length characters will be copied.

Possible ‘error_id’ values are defined in the gvd_def.h file.

---

short **GVD_get_version**(short mod_no, unsigned short * version);

Input parameters:

mod_no	module number (0 - 3)
*version	pointer to the version variable

Return value: 0 no errors, <0 error code (see gvd_def.h)

Description:

The procedure loads the 'version' variable with the FPGA version of the GVD module mod_no'. This is low a level procedure, not needed normally.

---

short **GVD_get_parameter**(short mod_no, short par_id, float * value);

Input parameters:

mod_no	module number (0 - 3)
par_id	parameter identification number (see gvd_def.h)
*value	pointer to the parameter value

Return value: 0 no errors, <0 error code (see gvd_def.h)

The procedure loads 'value' with the actual value of the requested parameter from the DLL-internal data structures of the GVD module 'mod_no'. The par_id values are defined in gvd_def.h file as GVD_PARAMETERS_KEYWORDS.

---

short **GVD_set_parameter**(short mod_no, short par_id, float value);

Input parameters:

mod_no	module number (0 - 3)
par_id	parameter identification number
value	new parameter value

Return value:

0 no errors, <0 error code (see gvd_def.h)

The procedure sets the specified hardware parameter. The value of the specified parameter is transferred to the internal data structures of the DLL functions and to the GVD module ‘mod_no’. The new parameter value is recalculated according to the parameter limits and hardware restrictions. Furthermore, cross dependencies between different parameters are taken into account to ensure the correct hardware operation. It is recommended to read back the parameters after setting to get their real values after recalculation.

The par_id values are defined in gvd_def.h file as GVD_PARAMETERS_KEYWORDS.

---

short **GVD_get_parameters**(short mod_no, GVDdata * data);

Input parameters:

mod_no	module number (0 - 3)
*data	pointer to result structure (type GVDdata)

Return value: 0 no errors, <0 error code (see gvd_def.h)

Description:

After calling the GVD_init function (see above) the measurement parameters from the initialisation file are present in the module and in the internal data structures of the DLLs. To give the user access to the parameters, the function **GVD_get_parameters** is provided. This function transfers the parameter values of the GVD module 'mod_no' from the internal structures of the DLLs into a structure of the type GVDdata (see gvd_def.h). A suitable structure has to be defined by the user. The parameter values in this structure are described below.

short active	most of the library functions are executed only when module is active (not 0), default 0
short frame_size	bits 0-3 - lg of frame size (quadratic frame) or lg of frame size X (rectangular frame) values in range: 4 (16x16) ... 12 (4096x4096), (default 9, 512x512) bits 8-11 - lg of frame size Y (rectangular frame) values in range: 4 (16) ... 14 (16384), (default 9, 512) bit 15 = 0 - quadratic frame, = 1 - rectangular frame
short lasers_active	bit 0(8) - laser 0(1) active, default 0 bit 1 - laser 0 & 1 off during flyback , default 0 bit 2 - = 0 - laser 0 is the first laser when multiplexing lasers = 1 - laser 1 is the first laser when multiplexing lasers , default 0, bit 2 valid for FPGA version > B2
unsigned short multiplex	lasers multiplexing mode (bits 0-1): 0 - off, 1 - within each pixel, 2 - after each line, 3 - after each frame , default 0 bits 2-15 - length of Laser1 phase in 1/100th of percent when mode = 1, default 5000 (50%) (first laser = laser 0 (FPGA version < B3) = bit 2 of laser_active (FPGA version > B2)

short limit_scan	0 - unlimited scan (default), 1 - scan specified no of frames 2 - repeated scan of specified no of frames (used with trigger)
unsigned short frame_counter	no of frames to scan when limit_scan > 0, default = 1
unsigned short scan_polarity	default 0 (all active low) bit 0 - pixel clock polarity, bit 1 - polarity of HSYNC (Line), bit 2 - polarity of VSYNC (Frame) bit = 0 - falling edge(active low) bit = 1 - rising edge(active high)
short scan_type	bit 0 = 0 scan frame (default), = 1 - scan one line only bit 3-1 line scanning type = 0 ramp continuous (default) (scans also during the pixel time), = 1 ramp steps (same position during the pixel) - increases min line time = 2 sinus continuous,
float line_time	line time in sec
float zoom_factor	zoom factor 1(default) .. 1024, max scan range (amplitude) is divided by the factor max depends on frame size: 4 for frames 2048x2048, 8 for frames 1024x1024, 16 for frames 512x512 32 for frames smaller than 512x512
float offset_x	offset X of the centre of zoomed area from the centre X of the total scan range, -100 % .. 100 % (default 0)
float offset_y	offset Y of the centre of zoomed area from the centre Y of the total scan range, -100 % .. 100 % (default 0)
float park_offs_x	offset X of the park position from the centre X of the total scan range, -100 % .. 100 % (default 0)
float park_offs_y	offset Y of the park position from the centre Y of the total scan range, -100 % .. 100 % (default 0)
float l1_power	Laser 1 power control 0 - 100%, default 50
float l2_power	Laser 2 power control 0 - 100%, default 50
float rect_zoom_x	X zoom factor for rectangular frame, 1(default) .. 1024,
float rect_zoom_y	Y zoom factor for rectangular frame, 1(default) .. 1024,
short scan_rate	0 - auto,scan rate fastest possible (default), 1 - scan rate defined by line_time
short park_center	bit 0 = 1 when parked, set beam position in the centre of scanned area, default 0 bit 4 (read only) shows current state of beam parking 1 - beam is parked, 0 - beam is not parked
short scan_trigger	External trigger condition (FPGA version > B0) bits 1 & 0 mean : 00 - (value 0) external trigger disabled (default), 01 - (value 1) external trigger active low, 10 - (value 2) external trigger active high

---

short **GVD_set_parameters** (short mod_no, GVDdata * data);

Input parameters:

mod_no	module number (0 - 3)
*data	pointer to parameters structure (type GVDdata, see gvd_def.h)

Return value: 0 no errors, <0 error code (see gvd_def.h)

Description:

The procedure sends all parameters from the ‘GVDdata’ structure to the internal DLL structures and to the control registers of the GVD module 'mod_no'.

The new parameter values are recalculated according to the parameter limits and hardware restrictions. Furthermore, cross dependencies between different parameters are taken into account to ensure the correct hardware operation. It is recommended to read back the parameters after setting to get their true values after recalculation. The values of ‘active’ is not changed. It can be changed only by a new ini_file in a GVD_init call or by GVD_set_mode call.

If an error occurs for a particular parameter, the procedure does not set the rest of the parameters and returns with an error code.

---

short **GVD_get_adjust_parameters** (short mod_no, GVD_Adjust_Para * adjpara, short from);

Input parameters:

mod_no	module number (0 - 3)
* adjpara	pointer to result structure
from	defines the source of adjust parameters: 0 - current adjust parameters, 1 - adjust parameters from EEPROM, 2 - default adjust parameters,

Return value: 0 no errors, <0 error code (see gvd_def.h)

The structure 'adjpara' is filled with adjust parameters taken from the source defined by 'from' parameter. The parameters can either be previously loaded from the EEPROM by GVD_init or GVD_get_eeprom_data or - not recommended - changed by GVD_set_adust_parameters.

The structure "GVD_Adjust_Para" is defined in the file gvd_def.h. It contains adjust and limit values used for calculating scan curve.

Normally, the adjust parameters need not be read explicitly because the EEPROM is read during GVD_init and the adjust values from EEPROM are loaded.

---

short **GVD_set_adjust_parameters** (short mod_no, GVD_Adjust_Para * adjpara, short save_to_eep);

Input parameters:

mod_no	module number (0 - 3)
* adjpara	pointer to a structure which contains new adjust parameters
save_to_eep	defines the source of adjust parameters: 0 - current adjust parameters, 1 - adjust parameters from EEPROM,
Return value:	0 no errors, <0 error code (see gvd_def.h)

The adjust parameters in the internal DLL structures of module 'mod_no' are set to values from the structure "adjpara" and, if 'save_to_eep' = 1, also to the EEPROM. The function is used to set adjust parameters of the module 'mod_no' to values other than the values from the EEPROM. We strongly discourage to use modified adjust parameters without consulting Becker & Hickl company, because they have a big influence on calculated scan curve and the module function can be seriously corrupted.

The structure "GVD_Adjust_Para" is defined in the file gvd_def.h.

---

short **GVD_get_eeprom_data**(short mod_no, GVD_EEP_Data *eep_data);

Input parameters:

mod_no	module number (0 - 3)
*eep_data	pointer to result structure
Return value:	0 no errors, <0 error code (see gvd_def.h)

The structure "eep_data" is filled with the contents of the EEPROM of the GVD module. The EEPROM contains the production data and adjust parameters of the module. The structure "GVD_EEP_Data" is defined in the file gvd_def.h.

---

short **GVD_write_eeprom_data**(short mod_no, unsigned short write_enable,
GVD_EEP_Data *eep_data);

Input parameters:

mod_no	module number (0 - 3)
write_enable	write enable password
*eep_data	pointer to a structure which will be sent to EEPROM

Return value: 0 no errors, <0 error code (see gvd_def.h)

The function is used to write data to the EEPROM of a GVD module 'mod_no' by the manufacturer. To prevent corruption of the data by not allowed access the function writes the EEPROM only if the ‘write_enable’ password is correct.

---

short **GVD_get_eeprom_defines** (short mod_no, GVD_EEP_Def *eep_def);

Input parameters:

mod_no	module number (0 - 3)
*eep_def	pointer to result structure

Return value: 0 no errors, <0 error code (see gvd_def.h)

The structure "eep_def" is filled with the contents of the EEPROM defines section of the GVD module 'mod_no'. The EEPROM defines contains currently the lasers names strings to be defined by the user. The structure "GVD_EEP_Def" is defined in the file gvd_def.h.

---

short **GVD_write_eeprom_defines** (short mod_no, GVD_EEP_Def *eep_def);

Input parameters:

mod_no	module number (0 - 3)
*eep_def	pointer to a structure which contains new EEPROM defines

Return value: 0 no errors, <0 error code (see gvd_def.h)

The function is used to write new EEPROM defines to the EEPROM of a GVD module 'mod_no'.

---

short **GVD_test_state** (short mod_no, short * state, short clear_flags);

Input parameters:

mod_no	module number (0 - 3)
*state	pointer to result value
clear_flags	0(1) clear resettable flags

Return value: 0 no errors, <0 error code (see gvd_def.h)

The **GVD_test_state** function returns the current status of the GVD module 'mod_no'. The status bits delivered by the function are listed below (see also gvd_def.h).

GVD_RUNNING	1	sequence is started
GVD_EOF_SEQ	2	sequence was finished
GVD_PARKED	0x10	beam is parked

For GVD-120 modules with FPGA version > B0 only

STATUS_WAIT_TRG	4	sequence waits for trigger
STATUS_SCAN	8	GVD delivers scan signals (when = 1)

The function is normally used to test whether the scan sequence is still running.

If 'clear_flags' parameter is set to 1, all resettable bits in status register are set to 0 (currently only GVD_EOF_SEQ bit).

---

short **GVD_start_scan**(short mod_no);

Input parameters:

mod_no	module number (0 - 3)

Return value: 0 no errors, <0 error code (see gvd_def.h)

The procedure is used to start sending the sequence of scan control signals to the GVD module 'mod_no' outputs. Depending on the parameter ‘LIMIT_SCAN’ the sequence runs continuously or it runs number of frames defined by 'FRAME_COUNTER'.
If the scan curve is not valid (due to changed parameters) the procedure prepares new scan curve. GVD-120 module checks trigger condition and if the trigger is enabled it will wait for trigger pulse before the start of scan sequence. (feature available on modules with FPGA v. B1 and newer).

The procedure **GVD_test_state** can be used to check whether the sequence was completed.

After the sequence was successfully completed **GVD_stop_scan** should be called to set the module to the initial state.

---

short **GVD_stop_scan**(short mod_no);

Input parameters:

mod_no module number (0 - 3)

Return value: 0 no errors, <0 error code (see gvd_def.h)

GVD_stop_scan is used to stop the running scan sequence on GVD module 'mod_no' by a software command. It must be also called after the finished scan sequence to set the module to the initial state

---

short **GVD_ prepare_scan_curve** (short mod_no);

Input parameters:

mod_no module number (0 - 3)

Return value: 0 no errors, <0 error code (see gvd_def.h)

Before scanning scan curve must be calculated and stored in GVD internal memory. This is done by **GVD_prepare_scan_curve** function. Changing most of GVD parameters causes the scan curve need to be recalculated. Therefore the function should be called after setting parameters directly before scan start.

---

short **GVD_park_beam** (short mod_no, short active, short center,

float * offs_x, float * offs_y);

Input parameters:

mod_no module number (0 - 3)

active	0 - switch off, 1 - switch on park beam
center	if = 1, park in the center of current scanning area
* offs_x	-100.. 100, pointer to new x offset (in percent) from the center of the current scanning area (used, when center = 0)
* offs_y	-100.. 100, pointer to new y offset (in percent) from the center of the current scanning area (used, when center = 0)

Return value: 0 no errors, <0 error code (see gvd_def.h)

The procedure is used to park beam in the specified position of the current scanned area for the GVD module 'mod_no' .

If 'active' = 0, the procedure switch off beam park and recalculates the scan curve.

If 'active' = 1, the procedure stops running scan sequence (if any),

the procedure parks beam to the specified position of the current scanning area:
if 'center' = 1, it is the center of scanning area

if 'center' = 0, it is the position defined by 'offs_x' and 'offs_y'

---

short **GVD_get_scan_info** (short mod_no, GVDScanInfo *scan_info);

Input parameters:

mod_no	module number (0 - 3)
*scan_info	pointer to result structure

Return value: 0 no errors, <0 error code (see gvd_def.h)

The structure "scan_info" is filled with the contents of the internal DLL structure which contains most important information about the current scan curve prepared for the GVD module 'mod_no'. The structure "GVDScanInfo" is defined in the file gvd_def.h.

---

short **DCS_init** (short mod_no);

Input parameters:

mod_no module number (0 - 3)

Return value:

0 no errors, <0 error code

Description:

DCS-BOX can be connected to a GVD module to simplify the cabling of the scanning system. DCS-BOX contains its own FPGA and EEPROM and it can be controlled by software.

DCS_init is called at the end of initialization of GVD module (GVD_init function)

DCS_init tests the communication between GVD module 'mod_no' and DCS_BOX connected to it (if any). If DCS_BOX is connected the **DCS_init** function:

- performs an EEPROM checksum test of DCS-BOX
- writes control parameters to DCS-BOX FPGA (parameters were set from a GVD ini_file during GVD_init call file or set to default if ini_file is NULL (not used)).

There is only one entry in ini_file which controls DCS-BOX

```
dcs_ctrl = 0x3c33          ; DCS box control word ( GVD FPGA version >= A7 )
                           ;   bits 0..7   low byte of control parameters
                           ;   bits 0..2 = SPC-A total no. of routing bits
                           ;   bit  3    = SPC-A laser routing enabled
                           ;   bits 4..6 = SPC-B total no. of routing bits
                           ;   bit  7    = SPC-B laser routing enabled
                           ;   bits 8..15   high byte of control parameters
                           ;   bit  8    = Red LED state if bit 9 = 1  - 0 = off, 1 = on
                           ;   bit  9    = 0 - Red LED of Scanner shows filter switch state
                           ;             = 1 - bit 8 controls red LED state
                           ;   bit  10   = SPC-A MARK 3 enable
                           ;   bit  11   = SPC-B MARK 3 enable
                           ;   bit  12   = SPC-A DCS-Switch signals OVLD to DCC
                           ;   bit  13   = SPC-B DCS-Switch signals OVLD to DCC
                           ;   bit  15   = always 1 when DCS is controlled by software
```

---

short **DCS_get_version**(short mod_no, unsigned char * version);

Input parameters:

mod_no	module number (0 - 3)
*version	pointer to the version variable

Return value: 0 no errors, <0 error code (see gvd_def.h)

Description:

The procedure loads the ‘version’ variable with the FPGA version of the DCS-BOX connected to GVD module 'mod_no’. This is low a level procedure, not needed normally.

---

short **DCS_get_init_status** (short mod_no, short * ini_status);

Input parameters:

mod_no	module number (0 - 3)
*ini_status	pointer to the initialisation status

Return value: 0 no errors, <0 error code (see gvd_def.h)

Description:

The procedure loads the ini_status variable with the initialisation result code set by the function DCS_init for the DCS-BOX connected to GVD module 'mod_no'. The possible values are shown below (see also gvd_def.h):

INIT_DCS_OK	0	initialization OK
INIT_DCS_WRONG_EEP_CHKSUM	-1	wrong DCS EEPROM checksum
INIT_DCS_NOT_DONE	-2	initialization not done
INIT_DCS_WRONG_GVD	-3	initialization not done, wrong FPGA version of GVD
INIT_DCS_NO_ACCESS_D0	-4	can't access DCS resources - data line hangs on 0
INIT_DCS_NO_ACCESS_D1	-5	can't access DCS resources - data line hangs on 1
INIT_DCS_EEP_ERR	-6	error when rd/wr DCS EEPROM
INIT_DCS_FPGA_ERR	-7	error when rd/wr DCS FPGA

---

short **DCS_get_parameters**(short mod_no, short from,DCSdata * data);

Input parameters:

mod_no	module number (0 - 3)
from	source of parameters : 0 - FPGA, 1 - DIP switches, 2 - DCS EEPROM
*data	pointer to result structure (type DCSdata)

Return value: 0 no errors, <0 error code (see gvd_def.h)

Description:

After calling the DCS_init function (see above) the control parameters from the initialisation file are sent to the DCS_BOX and in the internal data structures of the DLLs.

To give the user access to the parameters, the function **DCS_get_parameters** is provided. This function transfers the parameter values of the DCS-BOX connected to GVD module 'mod_no'. Depending on the parameter 'from' control parameters are taken from the DCS-

BOX FPGA, DIP switches or EEPROM into a structure of the type DCSdata (see gvd_def.h). A suitable structure has to be defined by the user. The parameter values in this structure are described below.

unsigned char ctrl_low	low byte of control parameters bits 0..2 = SPC-A total no. of routing bits bit 3 = SPC-A laser routing enabled bits 4..6 = SPC-B total no. of routing bits bit 7 = SPC-B laser routing enabled
unsigned char ctrl_high	high byte of control parameters bit 0 = Red LED state if bit 1 = 1 - 0 = off, 1 = on bit 1 = 0 - Red LED of Scanner shows filter switch state = 1 - bit 0 controls red LED state bit 2 = SPC-A MARK 3 enable bit 3 = SPC-B MARK 3 enable bit 4 = SPC-A DCS-Switch signals OVLD to DCC bit 5 = SPC-B DCS-Switch signals OVLD to DCC bit 7 = always 1 when DCS is controlled by software

---

short **DCS_set_parameters** (short mod_no, DCSdata * data);

Input parameters:

mod_no	module number (0 - 3)
*data	pointer to parameters structure (type DCSdata, see gvd_def.h)

Return value: 0 no errors, <0 error code (see gvd_def.h)

Description:

The procedure sends all parameters from the ‘DCSdata’ structure to the internal DLL structures and to FPGA and EEPROM of DCS-BOX connected to the GVD module 'mod_no'.

The new parameter values are recalculated according to the parameter limits and hardware restrictions.

---

short **DCS_read_status** (short mod_no, unsigned short * status);

Input parameters:

mod_no	module number (0 - 3)
*status	pointer to result value

Return value: 0 no errors, <0 error code (see gvd_def.h)

The **DCS_read_status** function returns the current status of the DCS-Box connected to GVD module 'mod_no'. The status bits delivered by the function are listed below (see also gvd_def.h).

DCS_SWITCH_0	1	switch 0 state
DCS_SWITCH_1	2	switch 1 state
SPC_A_NOT_830	4	A SPC module is not SPC-830
SPC_B_NOT_830	8	B SPC module is not SPC-830
SCANNING	0x10	scanning in progress

---

short **DCS_get_eeprom_data**(short mod_no, DCS_EEP_Data *eep_data);

Input parameters:

mod_no	module number (0 - 3)
*eep_data	pointer to result structure

Return value: 0 no errors, <0 error code (see gvd_def.h)

The structure "eep_data" is filled with the contents of the EEPROM of DCS-BOX connected to the GVD module 'mod_no'. The EEPROM contains the production data and adjust parameters of the module. The structure "DCS_EEP_Data" is defined in the file gvd_def.h.

---

short **DCS_write_eeprom_data**(short mod_no, unsigned short write_enable,
DCS_EEP_Data *eep_data);

Input parameters:

mod_no	module number (0 - 3)
write_enable	write enable password
*eep_data	pointer to a structure which will be sent to EEPROM

Return value: 0 no errors, <0 error code (see gvd_def.h)

The function is used to write data to the EEPROM of DCS-BOX connected to a GVD module 'mod_no' by the manufacturer. To prevent corruption of the data by not allowed access the function writes the EEPROM only if the ‘write_enable’ password is correct.

==========