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

MSA

Dynamic Link Libraries

User Manual

Version 3.0, Juni 2014

Introduction

The **MSA** Dynamic Link Library contains all functions to control the MSA 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 disks contain the following files:

MSA32.DLL	32-bit dynamic link library main file for use on 32-bit systems
MSA32x64.DLL	32-bit dynamic link library main file for use on 64-bit systems
MSA32.LIB	import library file for Microsoft Visual C/C++ for use on 32-bit systems
MSA32x64.LIB	import library file for Microsoft Visual C/C++ for use on 64-bit systems
MSA64.DLL	64-bit dynamic link library main file for use on 64-bit systems
MSA64.LIB	import library file for Microsoft Visual C/C++ for use on 64-bit systems
MSA_DEF.H	Include file containing Types definitions, Functions Prototypes and Pre-processor statements
MSA200.INI	MSA DLL initialisation file
MSADLL.DOC	This description file
USE_MSA.C	Simple example of using MSA DLL functions. Source file of the example is the file use_msa.c.

To install DLLs execute installation package (msa_setup_(32/64).exe) and follow its instructions.

MSA-DLL Functions list

The following functions are implemented in the MSA-DLL:

Initialisation functions:

MSA_init
MSA_test_if_active
MSA_get_init_status
MSA_get_mode
MSA_set_mode
MSA_get_version
MSA_get_module_info

Setup functions:

MSA_get_parameter
MSA_set_parameter
MSA_get_parameters
MSA_set_parameters
MSA_get_eeprom_data

MSA_write_eeprom_data
MSA_get_adjust_parameters
MSA_set_adjust_parameters

Status functions:

MSA_test_if_busy
MSA_read_status
MSA_get_current_sweep

Measurement control functions:

MSA_start_measure
MSA_stop_measure

MSA memory transfer functions:

MSA_fill_memory
MSA_read_data

Test functions:

MSA_test_id
MSA_get_test_error_string
MSA_get_error_string

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 'MSA', for example, MSAstd_test_id it is _stdcall version of MSA_test_id.
Description and behaviour of these functions are identical to the functions from the first (default) set – the only difference is calling convention.

Application Guide

Initialisation of the MSA Measurement Parameters

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 MSA module. This is accomplished by the function **MSA_init**.

The MSA DLL Functions can control up to four MSA modules on PCI bus (MSA-300, MSA-1000).

The **MSA_init** function

- reads the parameter values from a specified initialisation file
- checks the base I/O addresses for all active modules to avoid hardware conflicts
- checks and recalculates the parameters depending on the hardware restrictions and the adjust parameters from the EEPROM on each active MSA module
- sends the parameter values to the MSA control registers on each active MSA module
- performs a hardware test of each active MSA module

The initialisation file is an ASCII file with a structure shown in the table below. Each module has its own section in the initialisation file ([msa_module 0..3]). Only modules which have an

entry ‘active = 1’ are initialised. We recommend either to use the file MSA200.INI or to start with MSA200.INI and to introduce the desired changes.

```
;   MSA200 initialisation file
;   MSA parameters have to be included in .ini file only when parameter
;   value is different from default.
;   module section (msa_module0-3) is required for each existing MSA module

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

[msa_module0]                   ; MSA module 0 hardware parameters
base_adr = 0x380                ; base I/O address  for ISA module  (0 ... 0x3FC,default 0x380)
pci_card_no = 0                 ; number of module on PCI bus if PCI version of MSA module
                                ;   0 - 3, default -1 ( ISA module)
active = 1                      ;module active - can be used  (default = 0 - not active)
enable_meas = 1                 ; enable/disable(1/0) measurement , default = disable
trigger = 0                     ; external trigger condition
                                ;       none(0)(default),active low(1),active high(2)
inp_threshold = -0.1            ; input threshold level
                                ; (-0.5 ... 0.5V , default -0.1)
trig_threshold = -0.1           ; trigger threshold level
                                ; (-1.0 ... 1.0V , default -0.1)
time_per_point = 0.005          ; collection time of 1 point in mikrosec
                                ;  (0.005(default) ... 40.96), must be a multiple of time_resolution
                                ;         time_per_point must fulfil following expression:
                                ;     points_no * time_per_point[µs] <= max_point * time_resolution
                                ;     max_point = 131072 for MSA-1000, 524288 for MSA-200(300)
points_no = 1000                ; number of points to collect (64 ... max_point, default 1000)
                                ;          points_no must fulfil following expression:
                                ;     points_no * time_per_point[µs] <= max_point * time_resolution
                                ;     max_point = 131072 for MSA-1000, 524288 for MSA-200(300)
sweeps = 1                      ; number of accumulation sweeps (1(default) ... max_sweep)
                                ;     max_sweep = 65535 for MSA-200(300), 0xffffff for MSA-1000)
inp_holdoff = 0.5               ; input holdoff level  for MSA-1000
                                ; (0.5 ... 5.0 ns, default 0.5)
trig_holdoff = 0.5              ; trigger holdoff level  for MSA-1000
                                ; (0.5 ... 5.0 ns, default 0.5)
time_resolution = 0.005         ; time resolution  of 1 point in mikrosec
                                ;   0.005 for MSA200(300) ,
                                ;   0.001, 0.002, 0.004, 0.008(default) for MSA-1000
start_delay = 0.0               ; start delay in mikrosec for MSA-1000  0(default).. 1e6
active_edge= 0                  ; active edge of the input signal for MSA-1000  -
                                ;             0 - falling(default), 1 - rising

[msa_module1]                   ; MSA module 1 hardware parameters

base_adr = 0x280                ; base I/O address  for ISA module  (0 ... 0x3FC)
pci_card_no=1                   ; number of module on PCI bus if PCI version of MSA module
                                ;   0 - 3, default -1 ( ISA module)
active = 0                      ;module not active - cannot be used


[msa_module2]                   ; MSA module 2 hardware parameters

base_adr = 0x2a0                ;base I/O address  for ISA module  (0 ... 0x3FC)
```

```
pci_card_no=2          ; number of module on PCI bus if PCI version of MSA module
                       ;   0 - 3, default -1 ( ISA module)
active = 0             ;module not active - cannot be used


[msa_module3]          ; MSA module 3 hardware parameters

base_adr = 0x2c0       ;base I/O address  for ISA module  (0 ... 0x3FC)
pci_card_no=3          ; number of module on PCI bus if PCI version of MSA module
                       ;   0 - 3, default -1 ( ISA module)
active = 0             ;module not active - cannot be used
```

The module will be initialised, but only 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 MSA_set_mode with the parameter 'force_use' = 1.

After successful initialisation the module is locked to prevent that other application can access it (Locking modules does not apply to MSA-1000 module).

After an **MSA_init** call we recommend to call the **MSA_test_if_active** function to check which **MSA** modules are active. At least one module must be active, and only active modules can be operated further. It is recommended (but not required) to check also the initialisation status (by **MSA_get_init_status**) of each used module. In case of a wrong initialisation the initialisation status shows the reason of the error (see msa_def.h for possible values). In case of errors the function **MSA_get_error_string** returns error string and the function **MSA_get_test_error_string** returns additional information about the error in hardware test (value INIT_WRONG_DACS).

Additional information about MSA modules can be obtained by calling **MSA_get_module_info** function. The function fills MSAModInfo structure which is described below.

short module_type	module type : 20- MSA-200, 30 – MSA-300, 1000 – MSA-1000
short slot_number	slot number on PCI bus if MSA-300(1000) module
short in_use	-1 used and locked by other application, 0 - not used, 1 - in use (not for MSA-1000)
short init	set to initialisation result code
unsigned short base_adr	base I/O address

After calling the **MSA_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 **MSA_get_parameters** is provided. This function transfers the parameter values from the internal structures of the DLLs into a structure of the type MSAdata (see msa_def.h) which has to be declared by the user. The parameter values in this structure are described below.

unsigned short base_adr	lower 16-bit of base I/O address on PCI bus – cannot be changed
short init	set to initialisation result code
short active	most of the library functions are executed only when module is active (not 0)
short enable_meas	measurement enabled/disabled(1/0)
short test_eep	test EEPROM checksum on startup or not
short trigger	external trigger condition -

	none(0),active low(1),active high(2)
float inp_threshold	input threshold level (-0.5 ... 0.5V , default -0.1)
float trig_threshold	trigger threshold level (-1.0 ... 1.0V , default -0.1)
float time_per_point	collection time of 1 point in mikrosec (0.005(default) ... 40.96), must be a multiple of time_resolution time_per_point must fulfill following expression: points_no * time_per_point[µs] <= max_point * time_resolution max_point = 131072 for MSA-1000, 524288 for MSA-300
unsigned long points	no of points to collect (64 ... max_point, default 1000) points_no must fulfil following expression: points_no * time_per_point[µs] <= max_point * time_resolution max_point = 131072 for MSA-1000, 524288 for MSA-300
unsigned long sweeps	no of accumulation sweeps (1(default) ... 65535 (MSA-200,300), 0xffffffff (MSA-1000))
float inp_holdoff	input holdoff 0.5 .. 5.0[ns] , only MSA-1000
float trig_holdoff	trigger holdoff 0.5 .. 5.0[ns] , only MSA-1000
float time_resolution	time resolution of 1 point in mikrosec 0.005 for MSA200(300) , 0.001, 0.002, 0.004, 0.008(default) for MSA-1000
float start_delay	start delay after trigger in mikrosec for MSA-1000 0(default) .. 1e6
short pci_card_no	slot no for PCI module(0-3) or -1 for ISA module – cannot be changed
short active_edge	active edge of the input signal for MSA-1000 - 0 - falling(default), 1 - rising

To send the complete parameter set back to the DLLs and to the MSA module (e.g. after changing parameter values) the function **MSA_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 **MSA_set_parameters** by **MSA_get_parameters**.

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

Memory Read/Write Functions

Reading the memory of the MSA module is accomplished by the functions **MSA_read_data**. To fill the memory with a constant value (or to clear the memory) the function **MSA_fill_memory** is available.

Standard Measurements

The most important measurement functions are listed below.

The **MSA_test_if_busy** function is used to control the measurement loop. It sets a busy variable according to the current state of the measurement. The state of **all** active modules on which measurement is enabled is taken into account in the return value:

0 - all active MSA modules have finished the measurement,
1 - the measurement is still running (at least) in one MSA module, no modules are waiting for an external trigger

2 - at least one module is waiting for the external trigger of the first sweep
3 - at least one module is waiting for the external trigger of the current sweep

The **MSA_read_status** function returns the current status of a particular MSA module. The most important status bits delivered by the function are listed below (see also msa_def.h).

For MSA-200(300) modules

ARMED	0x8000	module is armed
ITRGED	0x4000	module was initially triggered
MEASURE	0x2000	module collects data (Armed and Triggered)
EOFM	0x1000	end of measurement
OVFL	0x800	overflow of one or more accumulators
TRGED	0x400	current sweep triggered

For MSA-1000 modules

ARMED1	0x40	module is armed
ITRGED1	0x8	module was initially triggered
MEASURE1	0x1	module collects data (Armed and Triggered)
EOFM1	0x2	end of measurement
OVFL1	0x20	overflow of one or more accumulators
TRGED1	0x10	current sweep triggered

MSA_start_measure starts the measurement in all active MSA modules. The measurement is controlled by the parameters loaded by the MSA_init, MSA_set_parameters or MSA_set_parameter functions. The recording procedure sweeps through the specified number of collection time bins ('points_no'). Subsequent sweeps are accumulated until the specified number of 'sweeps' are completed. To check whether a measurement is finished, the **MSA_test_if _busy** function is used.

A running measurement can be stopped by the **MSA_stop_measure** function.

In the figure below block diagram of a simple measurement routine is given.

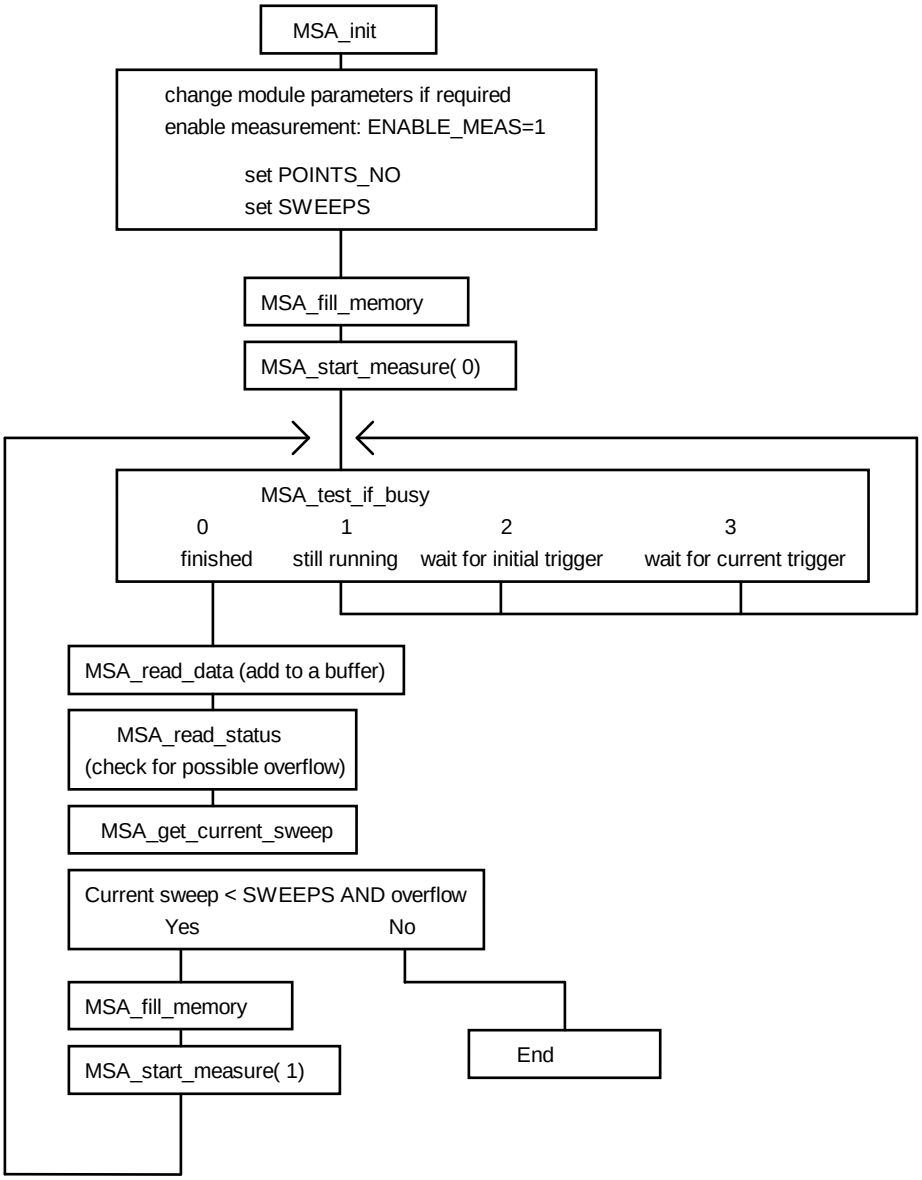

Fig1: Measurement with accumulation up to 65535 sweeps

Filling memory (MSA_fill_memory) is not required for MSA-1000 (done by hardware).

Error Handling

Each MSA 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 msa_def.h file and can be read using **MSA_get_error_string**. We recommend to check 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 MSA DLL Functions

---

short CVICDECL **MSA_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)

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 MSA module. This is accomplished by the function **MSA_init**. The function

- reads the parameter values from the specified file ini_file
- checks base I/O addresses for all active modules on ISA bus (MSA-200) to avoid hardware conflicts
- checks and recalculates the parameters depending on hardware constraints and adjust parameters from the EEPROM in each active MSA module
- sends the parameter values to the control registers of each active MSA module
- performs a hardware test of each active MSA module

The initialisation file is an ASCII file with a structure shown in the table below. We recommend either to use the file MSA200.INI or to start with MSA200.INI and introduce the desired changes.

```
;   MSA200 initialisation file
;   MSA parameters have to be included in .ini file only when parameter
;   value is different from default.
;   module section (msa_module0-3) is required for each existing MSA module

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

[msa_module0]                   ; MSA module 0 hardware parameters
base_adr = 0x380                ; base I/O address  for ISA module  (0 ... 0x3FC,default 0x380)
pci_card_no = 0                 ; number of module on PCI bus if PCI version of MSA module
                                ;   0 - 3, default -1 ( ISA module)
active = 1                      ;module active - can be used  (default = 0 - not active)
enable_meas = 1                 ; enable/disable(1/0) measurement , default = disable
trigger = 0                     ; external trigger condition
                                ;       none(0)(default),active low(1),active high(2)
inp_threshold = -0.1            ; input threshold level
                                ; (-0.5 ... 0.5V , default -0.1)
trig_threshold = -0.1           ; trigger threshold level
```

```
                        ; (-1.0 ... 1.0V , default -0.1)
time_per_point = 0.005  ; collection time of 1 point in mikrosec
                        ;  (0.005(default) ... 40.96), must be a multiple of time_resolution
                        ;        time_per_point must fulfil following expression:
                        ;     points_no * time_per_point[µs] <= max_point * time_resolution
                        ;     max_point = 131072 for MSA-1000, 524288 for MSA-200(300)
points_no = 1000        ; number of points to collect (64 ... max_point, default 1000)
                        ;         points_no must fulfil following expression:
                        ;     points_no * time_per_point[µs] <= max_point * time_resolution
                        ;     max_point = 131072 for MSA-1000, 524288 for MSA-200(300)
sweeps = 1              ; number of accumulation sweeps (1(default) ... max_sweep)
                        ;     max_sweep = 65535 for MSA-200(300), 0xffffff for MSA-1000)
inp_holdoff = 0.5       ; input holdoff level  for MSA-1000
                        ; (0.5 ... 5.0 ns, default 0.5)
trig_holdoff = 0.5      ; trigger holdoff level  for MSA-1000
                        ; (0.5 ... 5.0 ns, default 0.5)
time_resolution = 0.005 ; time resolution  of 1 point in mikrosec
                        ;   0.005 for MSA200(300) ,
                        ;   0.001, 0.002, 0.004, 0.008(default) for MSA-1000
start_delay = 0.0       ; start delay in mikrosec for MSA-1000  0(default).. 1e6
active_edge= 0          ; active edge of the input signal for MSA-1000  -
                        ;             0 - falling(default), 1 - rising

[msa_module1]           ; MSA module 1 hardware parameters

base_adr = 0x280        ; base I/O address  for ISA module  (0 ... 0x3FC)
pci_card_no=1           ; number of module on PCI bus if PCI version of MSA module
                        ;   0 - 3, default -1 ( ISA module)
active = 0              ;module not active - cannot be used


[msa_module2]           ; MSA module 2 hardware parameters

base_adr = 0x2a0        ;base I/O address  for ISA module  (0 ... 0x3FC)
pci_card_no=2           ; number of module on PCI bus if PCI version of MSA module
                        ;   0 - 3, default -1 ( ISA module)
active = 0              ;module not active - cannot be used


[msa_module3]           ; MSA module 3 hardware parameters

base_adr = 0x2c0        ;base I/O address  for ISA module  (0 ... 0x3FC)
pci_card_no=3           ; number of module on PCI bus if PCI version of MSA module
                        ;   0 - 3, default -1 ( ISA module)
active = 0              ;module not active - cannot be used
```

After an **MSA_init** call we recommend to call the **MSA_test_if_active** function and to check which MSA modules are active. Only active modules can be operated further, therefore at least one module must be active. It is reasonable also to check the initialisation status (**MSA_get_init_status**) of each used module. The initialisation status can show the reason of a wrong initialisation (see msa_def.h for possible values). In case of hardware test errors (values INIT_WRONG_COUNTER or INIT_WRONG_DACS) the function **MSA_get_test_error_string** delivers additional information.

Additional information about MSA modules can be obtained by calling **MSA_get_module_info** function. The function fills MSAModInfo structure (see msa_def.h for definition).

---

short CVICDECL **MSA_test_if_active** (short mod_no);

Input parameters:

mod_no module number (0 - 3)

Return value:

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

Description:

The procedure returns information whether the module specified by ‘mod_no’ is active or not. A module is set active only if there is the entry ‘active = 1’ in the respective module section in the ini_file. As a result of a wrong initialisation (MSA_init function) a module can be deactivated. To find out the reason of deactivating the module, run the MSA_get_init_status function.

---

short CVICDECL **MSA_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 msa_def.h)

Description:

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

INIT_OK	0	no error
INIT_NOT_DONE	-1	init not done
INIT_WRONG_EEP_CHKSUM	-2	wrong EEPROM checksum
INIT_WRONG_MOD_ID	-3	wrong module identification code
INIT_WRONG_BASE_ADR	-4	not unique base address
INIT_WRONG_DACS	-7	DAC’s test failed
INIT_CANT_OPEN_PCI_CARD	-8	cannot open PCI card
INIT_MOD_IN_USE	-9	module already in use
INIT_MSA_WINDRVR_VER	-10	incorrect WinDriver version
INIT_MSA_WRONG_LICENSE	-11	corrupted license key
INIT_MSA_NO_LICENSE	-12	license key not read from registry
INIT_MSA_LICENSE_NOT_VALID	-13	license is not valid for MSA DLL
INIT_MSA_LICENSE_DATE_EXP	-14	license date expired

In case of initialisation errors (values INIT_WRONG_DACS or INIT_CANT_OPEN_PCI_CARD) the function **MSA_get_test_error_string** gives additional information on the error.

---

short CVICDECL **MSA_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 msa_def.h file:

```
#define MSA_HARD        0       /* hardware mode */
#define MSA_SIMUL300    30      /* simulation mode of MSA-300 */
#define MSA_SIMUL1000   1000    /* simulation mode of MSA-1000 */
```

---

short CVICDECL **MSA_set_mode**(short mode, short force_use, short *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 msa_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 (not for MSA-1000) 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 MSA_get_init_status and/or MSA_get_module_info to check which modules are correctly initialised and can be use further.

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

---

short CVICDECL **MSA_get_version**(short mod_no , 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 msa_def.h)

Description:

The procedure loads the ‘version’ variable with the FPGA version of the module specified by mod_no. This is low a level procedure, not needed normally.

---

short CVICDECL **MSA_get_module_info** (short mod_no , MSAModInfo * 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 msa_def.h)

Description:

After calling the MSA_init function (see above) the MSAModInfo 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 MSAModInfo (see msa_def.h) which has to be defined by the user. The parameters included in this structure are described below.

short module_type	MSA module type (see msa_def.h)
short slot_number	slot number on PCI bus occupied by the MSA-300(1000) module
short in_use	-1 used and locked by other application, 0 - not used, 1 - in use (not for MSA-1000)
short init	set to initialisation result code
unsigned short base_adr	base I/O address

---

short CVICDECL **MSA_get_parameters**(short mod_no, MSAdata * data);

Input parameters:

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

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

Description:

After calling the MSA_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 **MSA_get_parameters** is provided. This function transfers the parameter values of the module 'mod_no' from the internal structures of the DLLs into a structure of the type MSAdata (see msa_def.h). A suitable structure has to be defined by the user. The parameter values in this structure are described below.

unsigned short base_adr	lower 16-bit of base I/O address on PCI bus – cannot be changed
short init	set to initialisation result code
short active	most of the library functions are executed only when module is active (not 0)
short enable_meas	measurement enabled/disabled(1/0)
short test_eep	test EEPROM checksum on startup or not
short trigger	external trigger condition - none(0),active low(1),active high(2)
float inp_threshold	input threshold level (-0.5 ... 0.5V , default -0.1)
float trig_threshold	trigger threshold level (-1.0 ... 1.0V , default -0.1)
float time_per_point	collection time of 1 point in mikrosec (0.005(default) ... 40.96), must be a multiple of time_resolution time_per_point must fulfill following expression: points_no * time_per_point[µs] <= max_point * time_resolution max_point = 131072 for MSA-1000, 524288 for MSA-300
unsigned long points	no of points to collect (64 ... max_point, default 1000) points_no must fulfil following expression: points_no * time_per_point[µs] <= max_point * time_resolution max_point = 131072 for MSA-1000, 524288 for MSA-300
unsigned long sweeps	no of accumulation sweeps (1(default) ... 65535 (MSA-200,300), 0xffffffff (MSA-1000))
float inp_holdoff	input holdoff 0.5 .. 5.0[ns] , only MSA-1000
float trig_holdoff	trigger holdoff 0.5 .. 5.0[ns] , only MSA-1000
float time_resolution	time resolution of 1 point in mikrosec 0.005 for MSA200(300) , 0.001, 0.002, 0.004, 0.008(default) for MSA-1000

float start_delay	start delay after trigger in mikrosec for MSA-1000 0(default) .. 1e6
short pci_card_no	slot no for PCI module(0-3) or -1 for ISA module – cannot be changed
short active_edge	active edge of the input signal for MSA-1000 - 0 - falling(default), 1 - rising

---

short CVICDECL **MSA_set_parameters**(short mod_no, MSAdata * data);

Input parameters:

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

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

Description:

The procedure sends all parameters from the 'MSAdata' structure to the internal DLL structures and to the control registers of the MSA module 'mod_no'.

The new parameter values are recalculated according to the parameter limits and hardware restrictions (e.g. DAC resolution). 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 'base_adr', 'init' and 'active' are not changed. They can be changed only by a new ini_file an a MSA_init 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 CVICDECL **MSA_get_parameter**(short mod_no, short par_id, float * value);

Input parameters:

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

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

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

---

short CVICDECL **MSA_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 msa_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 MSA module ‘mod_no’. The new parameter value is recalculated according to the parameter limits and hardware restrictions (e.g. DAC resolution). 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.

Parameters BASE_ADR and ACTIVE cannot be changed. They can be changed only by a new ini_file and a MSA_init call.

The par_id values are defined in msa_def.h file as MSA_PARAMETERS_KEYWORDS.

---

short CVICDECL **MSA_get_eeprom_data**(short mod_no, MSA_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 msa_def.h)

The structure "eep_data" is filled with the contents of the EEPROM of the MSA module specified by ‘mod_no’. The EEPROM contains the production data and the adjust parameters of the module. The structure "MSA_EEP_Data" is defined in the file msa_def.h.

Normally, the EEPROM data need not be read explicitly because the EEPROM is read during MSA_init and the module type information and the adjust values are taken into account when the MSA module registers are loaded.

---

short CVICDECL **MSA_write_eeprom_data**(short mod_no, unsigned short write_enable,
MSA_EEP_Data *eep_data);

Input parameters:

mod_no	module number (0 - 3)
write_enable	write enable password
*eep_data	pointer to result structure

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

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

---

short CVICDECL **MSA_get_adjust_parameters** (short mod_no, MSA_Adjust_Para *adjpara);

Input parameters:

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

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

The structure 'adjpara' is filled with adjust parameters of the MSA module ‘mod_no’ that are currently in use. The parameters can either be previously loaded from the EEPROM by MSA_init or MSA_get_eeprom_data or - not recommended - set by MSA_set_adust_parameters.

The structure "MSA_Adjust_Para" is defined in the file msa_def.h.

Normally, the adjust parameters need not be read explicitly because the EEPROM is read during MSA_init and the adjust values are taken into account when the MSA module registers are loaded.

---

short CVICDECL **MSA_set_adjust_parameters** (short mod_no, MSA_Adjust_Para *adjpara);

Input parameters:

mod_no	module number (0 - 3)
* adjpara	pointer to a structure which contains new adjust parameters

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

The adjust parameters in the internal DLL structures (not in the EEPROM) of the module ‘mod_no’ are set to values from the structure "adjpara". The function is used to set the module adjust parameters to values other than the values from the EEPROM. The new adjust values will be used until the next call of MSA_init. The next call to MSA_init replaces the adjust parameters by the values from the EEPROM. We strongly discourage to use modified adjust parameters, because the module function can be seriously corrupted by wrong adjust values.

The structure "MSA_Adjust_Para" is defined in the file msa_def.h.

---

short CVICDECL **MSA_test_if_busy**(short * busy);

Input parameters:

*busy pointer to result value

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

MSA_test_if_busy sets a ‘busy’ variable according to the current state of the measurement. The function is used to control the measurement loop after starting the measurement. Possible values of ‘busy’ are listed below.

0 - all active MSA modules finished the measurement,
1 - the measurement is still running at least in one MSA module, no modules are waiting for the trigger
2 - at least one module is waiting for the trigger for the first sweep
3 - at least one module is waiting for the trigger for the current sweep

---

short CVICDECL **MSA_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 msa_def.h)

The **MSA_read_status** function returns the current status of the MSA module defined by ‘mod_no’. The most important status bits delivered by the function are listed below (see also msa_def.h).

For MSA-200(300) modules

ARMED	0x8000	module is armed
ITRGED	0x4000	module was initially triggered
MEASURE	0x2000	module collects data (Armed and Triggered)

EOFM	0x1000	end of measurement
OVFL	0x800	overflow of one or more accumulators
TRGED	0x400	current sweep triggered

For MSA-1000 modules

ARMED1	0x40	module is armed
ITRGED1	0x8	module was initially triggered
MEASURE1	0x1	module collects data (Armed and Triggered)
EOFM1	0x2	end of measurement
OVFL1	0x20	overflow of one or more accumulators
TRGED1	0x10	current sweep triggered

The function is a low level procedure which is normally used only to test whether an overflow occurred during the measurement and to get additional information about the MSA module state. To control the measurement, the MSA_test_if_busy function is recommended.

---

short CVICDECL **MSA_get_current_sweep** (short mod_no, ,unsigned long *sweep);

Input parameters:

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

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

The **MSA_get_current_sweep** function fills ‘sweep’ with the current value of sweep counter in the MSA module ‘mod_no’.

The function is used to check how many sweeps were already collected during the measurement or when the measurement stops due to overflow.

---

short CVICDECL **MSA_start_measure**(short restart);

Input parameters:

restart	measurement restarted (1) or start from the beginning (0)

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

The procedure is used to start (restart) the measurement.

Before a measurement is started by **MSA_start_measure**

- the parameters on all active modules must be set (MSA_init or MSA_set_parameter(s)),
- the same measurement mode must be set for all requested modules,
- the measurement must be enabled in all requested modules (parameter ENABLE_MEAS must be set by MSA_set_parameter),

- MSA memory on all active modules must be cleared (from 0 to POINTS_NO –1) (not needed for MSA-1000)
- POINTS_NO and TIME_PER_POINT must be set to define the number of frames to be measured,
- SWEEPS must be set to define the number of sweeps to be accumulated,

The measurement continues until the specified number of points and accumulations has been reached or an overflow occurred.

In case of overflow the measurement can be restarted (if the current sweep counter is less than SWEEPS). If 'restart' is equal 1, sweeps counter is not reloaded and the current value is used. If restart is equal 0, the sweep counter is loaded with the 'SWEEPS' value.

---

short CVICDECL **MSA_stop_measure**(void);

Input parameters: none

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

MSA_stop_measure is used to stop the measurement by a software command.

---

short CVICDECL **MSA_fill_memory**(short mod_no, unsigned long from, unsigned long to, unsigned short fill_value);

Input parameters:

mod_no	module number (0 - 3)
from	1st address to fill (0 – POINTS_NO - 1)
to	last address to fill (from – POINTS_NO - 1)
fill value	value written to the MSA memory

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

The procedure is used to fill a specified part of the memory of the MSA module 'mod_no' with the value 'fill_value'.

---

short CVICDECL **MSA_read_data**(short mod_no, unsigned long from, unsigned long to, unsigned long * buf, short add);

Input parameters:

mod_no	module number (0 - 3)
from	1st address to read (0 – POINTS_NO - 1)

to	last address to read (from – POINTS_NO - 1)
*buf	pointer to data buffer to be filled
add	1 – add read values to the buffer, 0 – rewrite buffer contents

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

The procedure is used to read measurement results from the memory of the MSA module ‘mod_no’.

The procedure cannot be used during the measurement.

The procedure reads the MSA memory from the address ‘from’ up to the address ‘to’ and depending on the ‘add’ parameter either writes or add read values to the buffer ‘buf’ .

Using the procedure with the parameter ‘add’ equal 0 is recommended as a first call after the measurement start (then a previous clearing of the buffer is not needed).

A call with ‘add’ equal 1 is used to accumulate measurement results in the buffer when the measurement is restarted after an overflow.

Please make sure that the buffer ‘buf’ be allocated with enough memory for the required number of points (to - from +1).

---

short CVICDECL **MSA_test_id** (short mod_no) ;

Input parameters:

mod_no: module number (0 - 3)

Return value: on success - module type, on error <0 (error code)

The procedure is used to check the identification code of MSA module ‘mod_no’. It is a low level procedure that is called already during the initialisation by MSA_init. The procedure returns a module type value if the id is correct. Possible module type values are defined in the msa_def.h file.

---

short CVICDECL **MSA_get_test_error_string**(char *error_string);

Input parameters:

error_string pointer to error message string

Return value: <0 last error code (see msa_def.h)

The procedure fills ‘error_string’ with the internal DLL string generated during the last execution of the **MSA_init** function. ‘Error string’ contains detailed information on an initialisation error.

After a call to **MSA_get_test_error_string** DLL's internal error string is empty.

---

short CVICDECL **MSA_get_error_string**(short error_id, char * dest_string, short max_length);

Input parameters:

error_id	MSA DLL error id (0 – number of MSA errors-1) (see msa_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 MSA DLL error with id equal 'error_id'. Up to 'max_length characters will be copied.

Possible 'error_id' values are defined in the msa_def.h file.

==========