Becker & Hickl GmbH
Nahmitzer Damm
12277 Berlin
Tel. +49 30 787 56 32
Fax. +49 30 787 57 34
email: info@becker-hickl.de
http://www.becker-hickl.de

PMS-400

Dynamic Link Libraries

User Manual

Version 3.0, May 2012

Introduction

The PMS Dynamic Link Library contains all functions to control the PMS modules. The functions work under 32 or 64 bit Windows NT/2000/XP/7. 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:

PMS32.DLL	32-bit dynamic link library main file for use on 32-bit systems
PMS32x64.DLL	32-bit dynamic link library main file for use on 64-bit systems
PMS32.LIB	import library file for Microsoft Visual C/C++ for use on 32-bit systems
PMS32x64.LIB	import library file for Microsoft Visual C/C++ for use on 64-bit systems
PMS64.DLL	64-bit dynamic link library main file for use on 64-bit systems
PMS64.LIB	import library file for Microsoft Visual C/C++ for use on 64-bit systems
PMS_DEF.H	Include file containing Types definitions, Functions Prototypes and Pre-processor statements
PMS300.INI	DLL initialisation file for PMS modules
PMSDLL.DOC	This description file
USE_PMS.C	Simple example of using PMS DLL functions. Source file of the example is the file use_pms.c.

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

PMS-DLL Functions list

The following functions are implemented in the PMS-DLL:

Initialisation functions:

PMS_init
PMS_test_if_active
PMS_get_init_status
PMS_get_sync_adr
PMS_set_sync_adr
PMS_get_mode
PMS_set_mode
PMS_get_version
PMS_get_module_info

Setup functions:

PMS_get_parameter
PMS_set_parameter
PMS_get_parameters
PMS_set_parameters
PMS_get_eeprom_data
PMS_write_eeprom_data
PMS_get_adjust_parameters
PMS_set_adjust_parameters

Status functions:

PMS_test_if_busy
PMS_read_status
PMS_get_macro_time
PMS_get_points_no
PMS_get_sweep

Measurement control functions:

PMS_start_measure
PMS_stop_measure

PMS memory transfer functions:

PMS_fill_memory
PMS_read_data
PMS_read_events

Other functions:

PMS_test_id
PMS_test_counter
PMS_get_test_error_string
PMS_close

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 'PMS', for example, PMSstd_test_id it is _stdcall version of PMS_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 PMS 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 PMS module. This is accomplished by the function **PMS_init**.

The PMS DLL Functions can control up to eight PMS modules on PCI bus (PMS-400(A)). PMS-300 modules are not supported longer by the DLL (DLL v.2.51 is the last one with support for ISA bus modules).

PMS-400(A) modules cannot be synchronously started and stopped.

The **PMS_init** function

- checks whether DLL is correctly registered (looks for a BH license number and verifies it)
- reads the parameter values from a specified initialisation file
- checks and recalculates the parameters depending on the hardware restrictions and the adjust parameters from the EEPROM on each active PMS module
- sends the parameter values to the PMS control registers on each active PMS module
- performs a hardware test of each active PMS 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 ([pms_module0..7]). Only modules which have an entry ' active = 1' are initialised. We recommend either to use the file pms300.ini or to start with pms300.ini and to introduce the desired changes.

```
;    PMS300 initialisation file
;    PMS parameters have to be included in .ini file only when parameter
;    value is different from default.
;   module section ( pms_module0-7 ) is required for each existing PMS module ( up to 8 )
;   PMS main software  uses only module section pms_module0-3

[pms_base]
simulation = 0                  ; 0 - hardware mode(default) ,
                                ; >0 - simulation mode (see pms_def.h for possible values)
base_sync_adr =  0x398          ; sync_adr will be set on all active modules to start/stop
                                ; collection of photons (0 ... 0x3FC,default 0x398)
                                ; for ISA modules only

[pms_module0]                   ; PMS module 0 hardware parameters
base_adr = 0x380                ; base I/O address   (0 ... 0x3FC,default 0x380, ISA module)
active = 1                      ; module active - can be used  (default = 0 - not active)
pci_card_no= 0                  ; module number on PCI bus if PCI version of PMS module
                                ;   0 - 7, default -1 ( ISA module)
meas_mode = 0                   ; measurement mode, (0 - multiscaler (default), 1 - event)
enable_meas = 1                 ; enable/disable(1/0) measurement , default = enable
trigger = 0                     ; external trigger condition
                                ;       none(0)(default),active low(1),active high(2)
gate_level_A = 2.0              ; discriminator level for gate signal in channel A
                                ; ( -2.0V ... +2.0V , default 2.0)
gate_level_B = 2.0              ; discriminator level for gate signal in channel B
                                ;  ( -2.0V ... +2.0V , default 2.0)
inp_threshold_A = -0.1          ; input threshold level of channel A
                                ;  (-1.0 ... 1.0V , default -0.1)
inp_threshold_B = -0.1          ; input threshold level of channel B
                                ;  (-1.0 ... 1.0V , default -0.1)
event_threshold_A = 1           ; event threshold for channel A
                                ;  (1 ... 65535 , default 1)
event_threshold_B = 1           ;  event threshold for channel B
                                ;   (1 ... 65535 , default 1)
collect_time = 1.               ; collection time in seconds (default 1.0 sec)
                                ;  ( 0.25 mikrosec ... 100000 sec )
start_ptr = 0                   ; start point of collection( 0(default) ... 65535)
end_ptr = 0                     ; end  point of collection( start_ptr(default) ... 65535)
accumulate = 0                  ; accumulation of results in PMS memory on(1) / off(0 - default)
                                ;   used in multiscaler mode , for event mode always = 0
macro_time = 1.                 ; macro time (overall collection time of event mode measurement )
                                ; in seconds ( 0.25 mikrosec ... 10000000 sec, default 1000.sec )
```

```
trig_threshold = 2.0        ; trigger threshold level ( only for PCI version of PMS module )
                            ; (-2.0 ... 2.0 V , default 0.1 V)
sweeps = 0                  ;  no of accumulation sweeps for hardware accumulation
                            ;  = 0 for PMS-300 and PMS-400 with FPGA version < 0x301,
                            ; 0(default) ... 0xffffffff for
                            ;    PMS-400 with FPGA version >= 0x301,   PMS-400A
inp_holdoff = 15            ;  inputs holdoff 1..15(default),  only for PMS-400A
                            ;       defines holdoff time = 10.0 / inp_holdoff [ns]
out12v = 0                  ;  12V output active (1) / not active (0, default ),  only for PMS-400A


[pms_module1]               ; PMS module 1 hardware parameters

base_adr = 0x280            ; base I/O address   (0 ... 0x3FC, ISA module)
active = 0                  ; module not active - cannot be used
pci_card_no = 1             ; module number on PCI bus if PCI version of PMS module
                            ;   0 - 7, default -1 ( ISA module)
pci_bus_no = -1             ; PCI bus number on which PMS modules will be looking for
                            ;  ( important only when modules are on more than 1 busses )
                            ;   0 - 255, default -1 ( all PCI busses will be scanned)


[pms_module2]               ; PMS module 2 hardware parameters

base_adr = 0x2a0            ; base I/O address   (0 ... 0x3FC, ISA module)
active = 0                  ; module not active - cannot be used
pci_card_no = 2             ; module number on PCI bus if PCI version of PMS module
                            ;   0 - 7, default -1 ( ISA module)


[pms_module3]               ; PMS module 3 hardware parameters

base_adr = 0x2c0            ; base I/O address   (0 ... 0x3FC, ISA module)
active = 0                  ; module not active - cannot be used
pci_card_no = 3             ; module number on PCI bus if PCI version of PMS module
                            ;   0 - 7, default -1 ( ISA module)
```

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 PMS_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 **PMS_init** call we recommend to check which PMS modules are active by calling **PMS_test_if_active** function. 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 (**PMS_get_init_status**) of each used module. In case of a wrong initialisation the initialisation status shows the reason of the error (see pms_def.h for possible values). In case of hardware test errors (values INIT_WRONG_COUNTER or INIT_WRONG_DACS) the function **PMS_get_test_error_string** returns additional information about the error. Additional information about PMS modules can be obtained by calling **PMS_get_module_info** function. The function fills PMSModInfo structure which is described below.

```
short module_type           module type :  40 – PMS-400, 41 – PMS-400A
```

short bus_number	PCI bus number of the PMS-400(A) module
short slot_number	slot number on PCI bus if PMS-400(A) module
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	base I/O address - not valid in 64-bit operating systems

After calling the **PMS_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 **PMS_get_parameters** is provided. This function transfers the parameter values from the internal structures of the DLLs into a structure of the type PMSdata (see pms_def.h) which has to be defined by the user. The parameter values in this structure are described below.

short base_adr	I/O base address
short init	set to initialisation result code
short active	most of the library functions are executed only when module is active (not 0)
short test_eep	test EEPROM checksum on start-up or not
short meas_mode	measurement mode
short enable_meas	measurement enabled/disabled(1/0)
float gate_level_A	gate A discriminator level
float gate_level_B	gate B discriminator level
float inp_threshold_A	input threshold level of channel A
float inp_threshold_B	input threshold level of channel A
unsigned short event_threshold_A	event's threshold A
unsigned short event_threshold_B	event's threshold B
float collect_time	collection time interval
unsigned short start_ptr	memory start pointer
unsigned short end_ptr	memory end pointer
short accumulate	accumulate or overwrite values in memory
short trigger	external trigger condition - none(0),active low(1),active high(2)
float macro_time	macro collection time in event mode
float trig_threshold	trigger threshold level for PCI modules
unsigned long sweeps	no of accumulation sweeps 0 (PMS-300 and PMS-400 with FPGA version < 0x301), 0(default) ... 0xffffffff for PMS-400 with FPGA version >= 0x301 and PMS-400A
short pci_card_no	no of PCI module(0-7) or -1 for ISA module
unsigned short inp_holdoff	inputs holdoff 1..15(default), only for PMS-400A defines holdoff time = 10.0 / inp_holdoff [ns]
short out12v;	12V output active (1) / not active (0, default), only for PMS-400A

To send the complete parameter set back to the DLLs and to the PMS module (e.g. after changing parameter values) the function **PMS_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 **PMS_set_parameters** by **PMS_get_parameters**.

Single parameter values can be transferred to or from the DLL and module level by the functions **PMS_set_parameter** and **PMS_get_parameter**. To identify the desired parameter,

the parameter identification par_id is used. The parameter identification keywords are defined in pms_def.h.

Memory Configuration

The memory is organised in two banks which contain the data for channel A and B. In the multiscaler mode the channel data are 32 bit counter values. In the event mode a 32 bit counter value and a 32 bit macro time is stored for each event. Therefore, the measurement memory length is different depending on the measurement mode. In the multiscaler mode 65536 values can be stored for both channels. In the event mode 32768 events can be stored.

Memory Read/Write Functions

Reading the memory of the PMS module is accomplished by the functions **PMS_read_data** (for multiscaler mode) and **PMS_read_events** (for event mode). To fill the memory with a constant value (or to clear the memory) the function **PMS_fill_memory** is available.

Standard Measurements

The most important measurement functions are listed below.

The **PMS_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 is taken into account in the return value:

0 - all active PMS modules have finished the measurement,

1 - the measurement is still running (at least) in one PMS module, no modules are waiting for an external trigger

2 - at least one module is waiting for the external trigger or for external hardware arm

3 - at least in one module collection is disabled by external hardware

The **PMS_read_status** function returns the current status of a particular PMS module. The most important status bits delivered by the function are listed below (see also PMS_DEF.H).

ARMED	0x1	module is armed
MEASURE	0x2	module collects data (Armed and Triggered)
DIS_COL	0x40	start of next collection time interval is suppressed by external hardware
DIS_ARM	0x80	external hardware signals, that is not ready for arming
A_OVFL	0x100	overflow on counter A
B_OVFL	0x200	overflow on counter B

PMS_start_measure starts the measurement in all active PMS modules with the parameters set before by the PMS_init, PMS_set_parameters or PMS_set_parameter functions. The measurement starts to collect photons in subsequent points in PMS memory (or subsequent events in the event mode) from the address START_PTR. In the multiscaler mode the measurement stops when the memory pointer reaches END_PTR, i.e. after recording (End_ptr

-Start_ptr + 1) points. In the event mode the measurement stops when the memory pointer reaches END_PTR or after a time specified by MACRO_TIME .

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

In the figures below some block diagrams of some simple measurement routines are given.

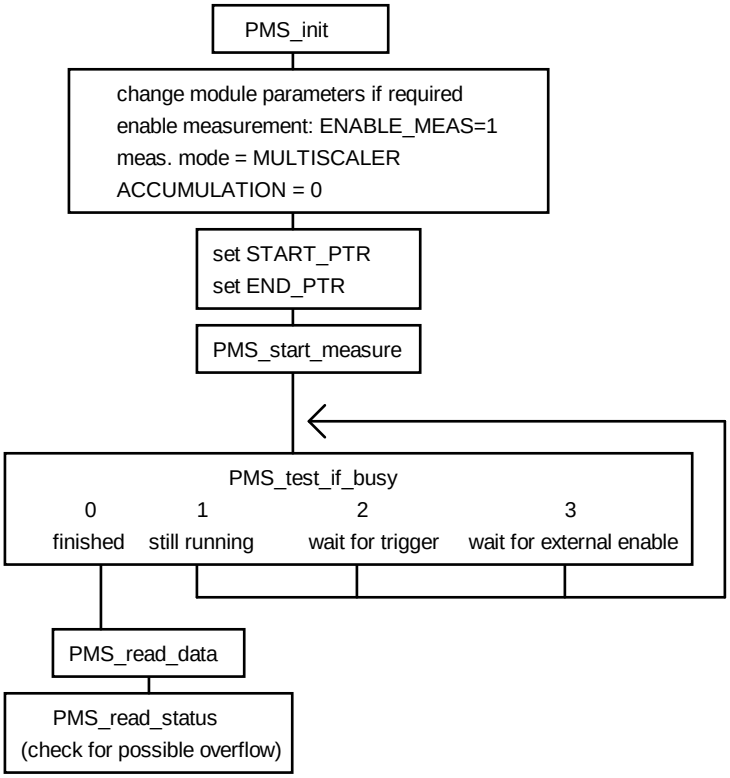

Fig1: Multiscaler measurement without accumulation

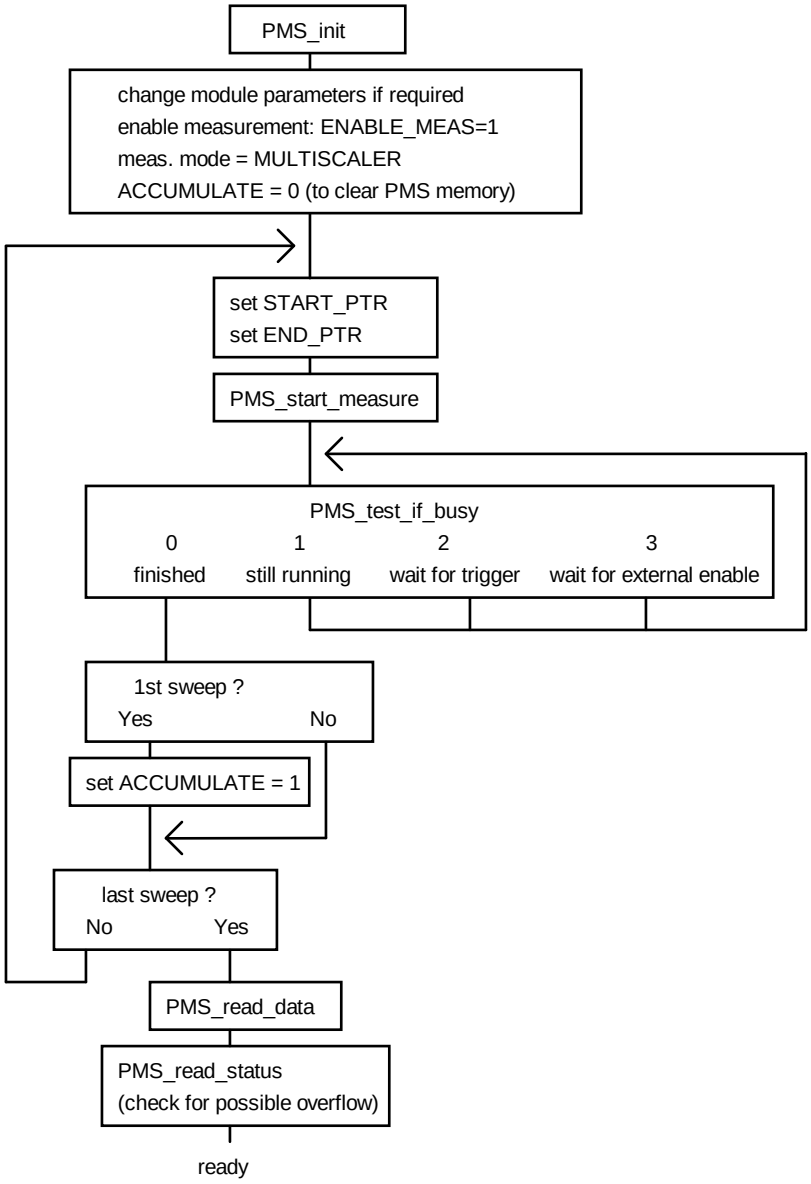

Fig. 2: Multiscaler Measurement with software accumulation
To clear the memory at the beginning of the measurement the first sweep is done with ACCUMULATE = 0.

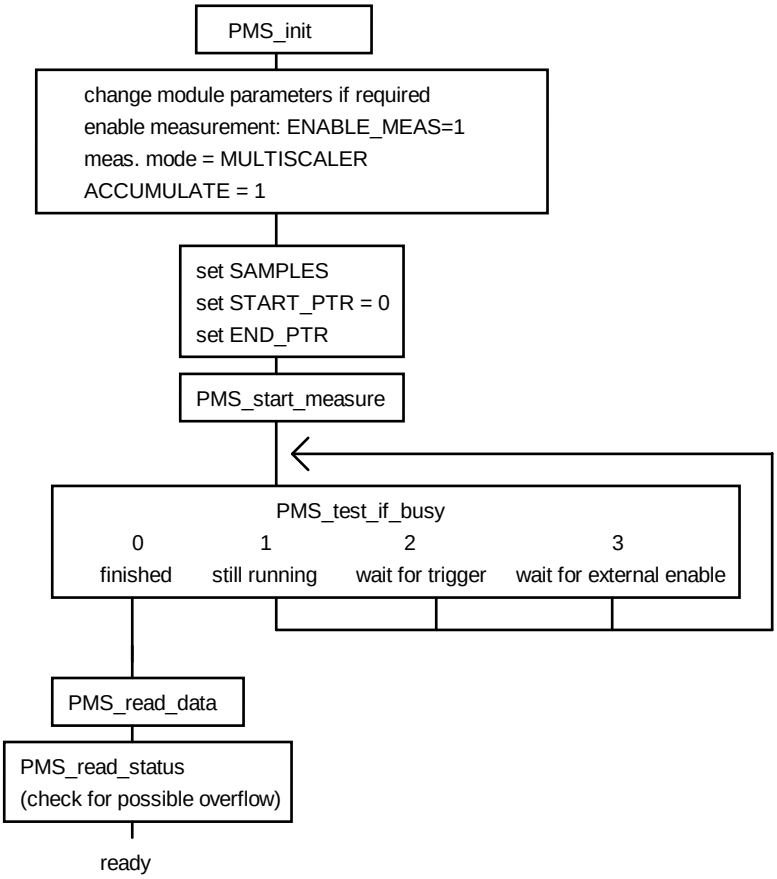

Fig. 3: Multiscaler Measurement with hardware accumulation

Hardware accumulation is possible when using PMS-400 modules with FPGA version >300(hex) and for PMS-400A. Use PMS_get_version function to check the FPGA version of your PMS-400 module.

START_PTR must be set to 0 (hardware requirement)

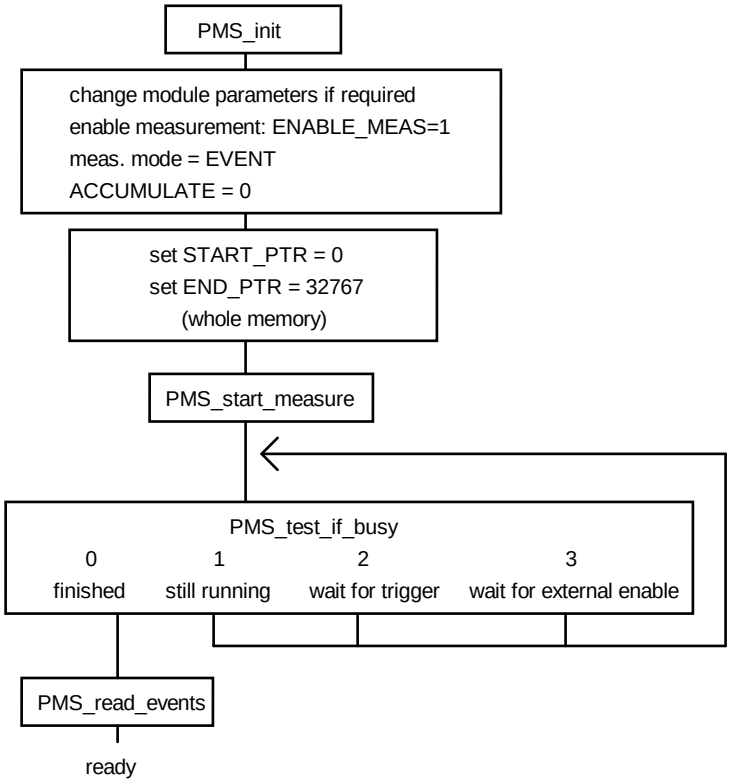

Fig. 4: Event mode measurement

Error Handling

Each PMS DLL function returns an error status. Return values >= 0 indicate error free execution. A value < 0 indicates that an error occurred during execution. The meaning of a particular error code can be found in pms_def.h file. 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)
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 PMS DLL Functions

---

short CVICDECL **PMS_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 PMS module. This is accomplished by the function **PMS_init**. The function

- checks whether DLL is correctly registered (looks for a BH license number and verifies it)
- reads the parameter values from the specified file ini_file
- checks and recalculates the parameters depending on hardware restrictions and adjust parameters from the EEPROM in each active PMS module
- sends the parameter values to the PMS control registers of each active PMS module
- performs a hardware test of each active PMS module

The initialisation file is an ASCII file with a structure shown in the table below. We recommend either to use the file pms300.ini or to start with pms300.ini and introduce the desired changes.

```
;   PMS300 initialisation file
;   PMS parameters have to be included in .ini file only when parameter
;   value is different from default.
;  module section ( pms_module0-7 ) is required for each existing PMS module ( up to 8 )
;  PMS main software  uses only module section pms_module0-3

[pms_base]
simulation = 0                  ; 0 - hardware mode(default) ,
                                ; >0 - simulation mode (see pms_def.h for possible values)
base_sync_adr =  0x398          ; sync_adr will be set on all active modules to start/stop
                                ; collection of photons (0 ... 0x3FC,default 0x398)
                                ; for ISA modules only

[pms_module0]                   ; PMS module 0 hardware parameters
base_adr = 0x380                ; base I/O address   (0 ... 0x3FC,default 0x380, ISA module)
active = 1                      ; module active - can be used  (default = 0 - not active)
pci_card_no = 0                 ; module number on PCI bus if PCI version of PMS module
                                ;   0 - 7, default -1 ( ISA module)
meas_mode = 0                   ; measurement mode, (0 - multiscaler (default), 1 - event)
enable_meas = 1                 ; enable/disable(1/0) measurement , default = enable
trigger = 0                     ; external trigger condition
```

```
                                ;       none(0)(default), active low(1), active high(2)
gate_level_A= 2.0               ; discriminator level for gate signal in channel A
                                ;  ( -2.0V ... +2.0V , default 2.0)
gate_level_B= 2.0               ; discriminator level for gate signal in channel B
                                ;  ( -2.0V ... +2.0V , default 2.0)
inp_threshold_A= -0.1           ; input threshold level of channel A
                                ;  (-1.0 ... 1.0V , default -0.1)
inp_threshold_B= -0.1           ; input threshold level of channel B
                                ;  (-1.0 ... 1.0V , default -0.1)
event_threshold_A = 1           ; event threshold for channel A
                                ;  (1 ... 65535 , default 1)
event_threshold_B = 1           ; event threshold for channel B
                                ;  (1 ... 65535 , default 1)
collect_time= 1.                ; collection time in seconds (default 1.0 sec)
                                ;   ( 0.25 mikrosec ... 100000 sec )
start_ptr =0                    ; start point of collection( 0(default) ... 65535)
end_ptr =0                      ; end  point of collection( start_ptr(default) ... 65535)
accumulate =0                   ; accumulation of results in PMS memory on(1) / off(0 - default)
                                ;   used in multiscaler mode , for event mode always = 0
macro_time = 1.                 ; macro time (overall collection time of event mode measurement )
                                ; in seconds ( 0.25 mikrosec ... 10000000 sec, default 1000.sec )
trig_threshold = 2.0            ; trigger threshold level ( only for PCI version of PMS module )
                                ; (-2.0 ... 2.0 V , default 0.1 V)
sweeps = 0                      ;  no of accumulation sweeps for hardware accumulation
                                ;  = 0 for PMS-300 and PMS-400 with FPGA version < 0x301,
                                ;  0(default) ... 0xffffffff for
                                ;    PMS-400 with FPGA version >= 0x301,   PMS-400A
inp_holdoff = 15                ;  inputs holdoff 1..15(default),  only for PMS-400A
                                ;       defines holdoff time = 10.0 / inp_holdoff [ns]
out12v = 0                      ;  12V output active (1) / not active (0, default ),  only for PMS-400A


[pms_module1]                   ; PMS module 1 hardware parameters

base_adr = 0x280                ; base I/O address   (0 ... 0x3FC, ISA module)
active = 0                      ; module not active - cannot be used
pci_card_no = 1                 ; module number on PCI bus if PCI version of PMS module
                                ;   0 - 7, default -1 ( ISA module)
pci_bus_no = -1                 ; PCI bus number on which PMS modules will be looking for
                                ;  ( important only when modules are on more than 1 busses )
                                ;   0 - 255, default -1 ( all PCI busses will be scanned)


[pms_module2]                   ; PMS module 2 hardware parameters

base_adr = 0x2a0                ; base I/O address   (0 ... 0x3FC, ISA module)
active = 0                      ; module not active - cannot be used
pci_card_no = 2                 ; module number on PCI bus if PCI version of PMS module
                                ;   0 - 7, default -1 ( ISA module)


[pms_module3]                   ; PMS module 3 hardware parameters

base_adr = 0x2c0                ; base I/O address   (0 ... 0x3FC, ISA module)
active = 0                      ; module not active - cannot be used
pci_card_no = 3                 ; module number on PCI bus if PCI version of PMS module
                                ;   0 - 7, default -1 ( ISA module)
```

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 PMS_set_mode with the parameter ‘force_use’ = 1.

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

After **PMS_init** call we recommend to check which PMS modules are active by calling **PMS_test_if_active** function (minimum one module must be active and only active modules can be operated further). It is reasonable also to check the initialisation status (**PMS_get_init_status**) of each used module. The initialisation status can show the reason of a wrong initialisation (see pms_def.h for possible values). In case of hardware test errors (values INIT_WRONG_COUNTER or INIT_WRONG_DACS) the function **PMS_get_test_error_string** delivers additional information.

Additional information about PMS modules can be obtained by calling **PMS_get_module_info** function. The function fills PMSModInfo structure (see pms_def.h for definition).

---

short CVICDECL **PMS_test_if_active** (short mod_no);

Input parameters:

mod_no module number (0 - 7)

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 (PMS_init function) a module can be deactivated. In this case a the PMS_get_init_status function can explain the reason of deactivating the module.

---

short CVICDECL **PMS_get_init_status**(short mod_no, short * ini_status);

Input parameters:

mod_no	module number (0 - 7)
*ini_status	pointer to the initialisation status

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

Description:

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

INIT_PMS_OK	0	no error
INIT_ PMS_NOT_DONE	-1	init not done
INIT_ PMS_WRONG_EEP_CHKSUM	-2	wrong EEPROM checksum
INIT_ PMS_WRONG_MOD_ID	-3	wrong module identification code
INIT_ PMS_WRONG_BASE_ADR	-4	not unique base address
INIT_ PMS_WRONG_SYNC_ADR	-5	sync address equal to base address
INIT_ PMS_WRONG_COUNTER	-6	counter test failed
INIT_ PMS_WRONG_DACS	-7	DAC’s test failed
INIT_ PMS_CANT_OPEN_PCI_CARD	-8	cannot open PCI card
INIT_ PMS_MOD_IN_USE	-9	module already in use
INIT_ PMS_WINDRVR_VER	-10	incorrect WinDriver version
INIT_ PMS_WRONG_LICENSE	-11	corrupted license key
INIT_PMS_NO_LICENSE	-12	license key not read from registry
INIT_PMS_LICENSE_NOT_VALID	-13	license is not valid for PMS DLL
INIT_PMS_LICENSE_DATE_EXP	-14	license date expired

In case of hardware test errors (values INIT_WRONG_COUNTER or INIT_WRONG_DACS) the function **PMS_get_test_error_string** gives additional information on the test error.

---

short CVICDECL **PMS_get_sync_adr**(short * adr);

Input parameters:

*adr pointer to the sync address

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

Description:

The procedure loads the ‘adr’ variable with the actual sync address value.

The Sync address is common for all active PMS-300 modules and is used to synchronously start and stop all PMS-300 modules. The procedure returns an error, when the sync address has not the same value for all modules.

---

short CVICDECL **PMS_set_sync_adr**(short adr);

Input parameters:

mod_no module number (0 - 7)
adr new value of sync address

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

Description:

The procedure sets the sync address value to ‘adr’.

The Sync address is common for all active PMS-300 modules and is used to synchronously start/stop all PMS-300 modules. The procedure checks for conflicts with I/O addresses used by the active PMS modules and, - if there are no conflicts- changes the sync address.

The procedure should be used only if the actual sync address (set during the initialisation to a value taken from ini_file) causes conflicts with other devices.

---

short CVICDECL **PMS_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 pms_def.h file:

```
#define PMS_HARD        0       /* hardware mode */
#define PMS_SIMUL300    30      /* simulation mode of PMS-300 */
#define PMS_SIMUL400    40      /* simulation mode of PMS-400 */
#define PMS_SIMUL400A   41      /* simulation mode of PMS-400A */
```

---

short CVICDECL **PMS_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 pms_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 8 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 8 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 cannot be used further.

When one of the simulation modes is requested for each of 8 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 cannot be used further.

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

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

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

---

short CVICDECL **PMS_get_version**(short mod_no , short * version);

Input parameters:

mod_no	module number (0 - 7)
*version	pointer to the version variable

Return value: 0 no errors, <0 error code (see pms_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 **PMS_get_module_info** (short mod_no , PMSModInfo * mod_info);

Input parameters:

mod_no	module number (0 - 7)
* mod_info	pointer to the result structure

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

Description:

After calling the PMS_init function (see above) the PMSModInfo internal structures for all 8 modules are filled. This function transfers the contents of the internal structure of the DLL into a structure of the type PMSModInfo (see pms_def.h) which has to be defined by the user. The parameters included in this structure are described below.

short module_type	PMS module type (see pms_def.h)
short bus_number	PCI bus number of the PMS-400(A) module
short slot_number	slot number on PCI bus occupied by the PMS-400(A) module
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	base I/O address - not valid in 64-bit operating systems

---

short CVICDECL **PMS_get_parameters**(short mod_no, PMSdata * data);

Input parameters:

mod_no	module number (0 - 7)
*data	pointer to result structure (type PMSdata)

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

Description:

After calling the PMS_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 **PMS_get_parameters** is provided. This function transfers the parameter values from the DLLs internal structures of the module 'mod_no' into a structure of the type PMSdata (see pms_def.h) which has to be defined by the user. The parameter values in this structure are described below.

short base_adr	I/O base address
short init	set to initialisation result code
short active	most of the library functions are executed only when module is active (not 0)
short test_eep	test EEPROM checksum on start-up or not
short meas_mode	measurement mode
short enable_meas	measurement enabled/disabled(1/0)
float gate_level_A	gate A discriminator level
float gate_level_B	gate B discriminator level
float inp_threshold_A	input threshold level of channel A
float inp_threshold_B	input threshold level of channel A
unsigned short event_threshold_A	event's threshold A
unsigned short event_threshold_B	event's threshold B
float collect_time	collection time interval
unsigned short start_ptr	memory start pointer
unsigned short end_ptr	memory end pointer
short accumulate	accumulate or overwrite values in memory
short trigger	external trigger condition - none(0),active low(1),active high(2)
float macro_time	macro collection time in event mode
float trig_threshold	trigger threshold level for PCI modules
unsigned long sweeps	no of accumulation sweeps 0 (PMS-300 and PMS-400 with FPGA version < 0x301),

	0(default) ... 0xffffffff for PMS-400 with FPGA version >= 0x301 and PMS-400A
short pci_card_no	no of PCI module(0-7) or -1 for ISA module
unsigned short inp_holdoff	inputs holdoff 1..15(default), only for PMS-400A defines holdoff time = 10.0 / inp_holdoff [ns]
short out12v;	12V output active (1) / not active (0, default), only for PMS-400A

---

short CVICDECL **PMS_set_parameters**(short mod_no, PMSdata * data);

Input parameters:

mod_no	module number (0 - 7)
*data	pointer to parameters structure (type PMSdata, see pms_def.h)

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

Description:

The procedure sends all parameters from the 'PMSdata' structure to the internal DLL structures and to the control registers of the PMS 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 real 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 PMS_init call.

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

---

short CVICDECL **PMS_get_parameter**(short mod_no, short par_id, float * value);

Input parameters:

mod_no	module number (0 - 7)
par_id	parameter identification number (see pms_def.h)
*value	pointer to the parameter value

Return value: 0 no errors, <0 error code (see pms_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 pms_def.h file as PMS_PARAMETERS_KEYWORDS.

---

short CVICDECL **PMS_set_parameter**(short mod_no, short par_id, float value);

Input parameters:

mod_no	module number (0 - 7)
par_id	parameter identification number
value	new parameter value

Return value:

0 no errors, <0 error code (see pms_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 PMS 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 new ini_file and a PMS_init call.

The par_id values are defined in pms_def.h file as PMS_PARAMETERS_KEYWORDS.

---

short CVICDECL **PMS_get_eeprom_data**(short mod_no, PMS_EEP_Data *eep_data);

Input parameters:

mod_no	module number (0 - 7)
*eep_data	pointer to result structure

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

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

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

---

short CVICDECL **PMS_write_eeprom_data**(short mod_no, unsigned short write_enable,
PMS_EEP_Data *eep_data);

Input parameters:

mod_no	module number (0 - 7)
write_enable	write enable pasword
*eep_data	pointer to result structure

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

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

---

short CVICDECL **PMS_get_adjust_parameters** (short mod_no, PMS_Adjust_Para *adjpara);

Input parameters:

mod_no	module number (0 - 7)
* adjpara	pointer to result structure

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

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

The structure "PMS_Adjust_Para" is defined in the file pms_def.h.

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

---

short CVICDECL **PMS_set_adjust_parameters** (short mod_no, PMS_Adjust_Para *adjpara);

Input parameters:

mod_no	module number (0 - 7)
* adjpara	pointer to a structure which contains new adjust parameters

Return value: 0 no errors, <0 error code (see pms_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 PMS_init. The next call to PMS_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 "PMS_Adjust_Para" is defined in the file pms_def.h.

---

short CVICDECL **PMS_test_if_busy**(short * busy);

Input parameters:

*busy pointer to result value

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

PMS_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. The current state of all active modules is taken into account. Possible values of busy are listed below.

0 - all active PMS modules finished the measurement,

1 - the measurement is still running at least in one PMS module, no modules are waiting for the trigger

2 - at least one module is waiting for the trigger

3 - at least one module is waiting for enable by external hardware

---

short CVICDECL **PMS_read_status**(short mod_no, unsigned short * status);

Input parameters:

mod_no module number (0 - 7)
*status pointer to result value

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

The **PMS_read_status** function returns the current status of PMS module ‘mod_no’. The most important status bits delivered by the function are listed below (see also PMS_DEF.H).

ARMED	0x1	module is armed
MEASURE	0x2	module collects data (Armed and Triggered)
DIS_COL	0x40	start of next collection time interval is suppressed by external hardware
DIS_ARM	0x80	external hardware signals, that is not ready for arming
A_OVFL	0x100	overflow on counter A

B_OVFL	0x200	overflow on counter B

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 PMS module state.

To control the measurement, the PMS_test_if_busy function is recommended. If the module is not armed you can check the reason (not triggered or external hardware disabled).

---

short CVICDECL **PMS_get_macro_time**(short mod_no, double * time);

Input parameters:

mod_no	module number (0 - 7)
*time	pointer to result value

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

The **PMS_get_macro_time** function fills ‘time’ with the current value (in seconds) of the macro time in the PMS module ‘mod_no’. The macro time is valid for event mode measurements only and expresses the total time from the start of the photon collection.

The function is used to check the progress of event mode measurement. Remember that macro time is internally stored as multiples of collection time. Thus, changing the parameter COLLECT_TIME results in a different time value delivered by the next PMS_get_macro_time call.

---

short CVICDECL **PMS_get_points_no**(short mod_no, unsigned long * points_no);

Input parameters:

mod_no	module number (0 - 7)
* points_no	pointer to result value

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

The **PMS_get_points_no** function sets the ‘points_no’ variable to the current number of points(events) collected from the start of the last measurement in the PMS module ‘mod_no’. Points_no is set as a difference between the current value of the memory pointer and the start pointer.

The function is used to check how many points (events) were already collected during the measurement.

---

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

Input parameters:

mod_no	module number (0 - 7)
*sweep	pointer to result value

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

The **PMS_get_sweep** function sets the 'sweep' variable to the current number of sweeps accumulated from the start of the last measurement (with hardware accumulation) in the PMS module 'mod_no'.

Hardware accumulation is possible when using PMS-400A and PMS-400 modules with FPGA version > 300(hex). Use PMS_get_version function to check the FPGA version of your PMS module.

The function is used to check how many sweeps were already accumulated during the measurement with hardware accumulation.

---

short CVICDECL **PMS_start_measure**(void);

Input parameters: none

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

The procedure is used to start the measurement.

Before a measurement is started by **PMS_start_measure**

- the parameters on all active modules must be set (PMS_init or PMS_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 PMS_set_parameter),
- START_PTR and END_PTR must be set to define the number of points to be measured,
- for event mode MACRO_TIME must be to define the maximum overall measurement time.

In the event mode the macro time clock and the PMS memory (from START_PTR to END_PTR) are cleared when the measurement starts.

In the multiscaler mode the measurement stops the when the memory pointer reaches END_PTR, i.e. after collecting End_ptr -Start_ptr + 1 points. In the event mode the measurement stops when the memory pointer reaches END_PTR or after a time specified by MACRO_TIME (the 1st stop condition stops measurement).

---

short CVICDECL **PMS_stop_measure**(void);

Input parameters: none

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

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

---

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

Input parameters:

mod_no	module number (0 - 7)
channel	channel A: 0 , channel B: 1 , both channels: -1
from	1st address to fill (0 - 65535(32767 for event mode))
to	last address to fill (from - 65535(32767 for event mode))
fill value	value written to the PMS memory

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

The procedure is used to fill a specified part of the memory of the PMS module ‘mod_no’ with the value ‘fill_value’.

In the event mode and in the multiscaler mode with ACCUMULATE=0 memory clearing is not required.

---

short CVICDECL **PMS_read_data**(short mod_no, unsigned long * points_read, unsigned short from, unsigned short to, unsigned long * A_buf, unsigned long * B_buf);

Input parameters:

mod_no	module number (0 - 7)
* points_read	pointer to variable which will be set with number of read points
from	1st address to read (0 - 65535)
to	last address to read (from - 65535)
* A_buf	pointer to data buffer to be filled with channel A data
* B_buf	pointer to data buffer to be filled with channel B data

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

The procedure is used to read **multiscaler mode measurement results** from the PMS module ‘mod_no’. To read results obtained in the event mode, use the PMS_read_events procedure.

The number of points read (32 bit counter values) depends on the state of the measurement.
If the measurement is already finished (or not started yet), the procedure reads the PMS memory from the address ‘from’ up to the address ‘to’ and sets the points_read variable to the value ‘ to - from +1’.

If the measurement is still running, the procedure reads the PMS memory from the address ‘from’ up to the current memory pointer. Then it reads the last (not finished) point value directly from the counter. Finally, the procedure sets the ‘points_read’ variable to a value equal to the number of points collected from the start of the measurement.

On the PMS-400A module Bus Master DMA transfer is used to ensure very fast data readout.

The reading of an unused channel can be suppressed by setting the corresponding A(B)_buf parameter to NULL.

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

```
short CVICDECL PMS_read_events(short mod_no, unsigned long * events_read,
                               unsigned short from, unsigned short to,
                               unsigned long * A_data, unsigned long * B_data,
                               float  * A_time, float * B_time);
```

Input parameters:

mod_no	module number (0 - 7)
* events_read	pointer to variable which will be set with number of read events
from	1st address to read (0 - 32767)
to	last address to read (from - 32767)
* A_data	pointer to buffer to be filled with counter values of channel A events
* B_data	pointer to buffer to be filled with counter values of channel B events
* A_time	pointer to buffer to be filled with macro times of channel A events
* B_time	pointer to buffer to be filled with macro times of channel B events

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

The procedure is used to read **event mode measurement results** from the PMS module ‘mod_no’ memory. To read results obtained in the multiscaler mode, use the PMS_read_data procedure.

The parameter ‘from’ should be in the range from START_PTR to (START_PTR + number of collected events). The number of collected events can be checked by the function PMS_get_points_no.
The variable ‘events_read’ is filled with the number of events which were read (this value can be less than (to- from), if there are less events in the memory).

For the individual events, the counter values are put to A(B)_data buffers and the macro time values (in seconds) are put into the A(B)_time buffers.
On the PMS-400A module Bus Master DMA transfer is used to ensure very fast data readout.

The reading of an unused channel can be suppressed by setting the corresponding A(B)_data or A(B)_time parameter to NULL.
Please make sure that the A(B)_data and the A(B)_time buffers be allocated with enough memory for (to - from +1) 32 bit values.

---

short CVICDECL **PMS_test_id**(short mod_no) ;

Input parameters:

mod_no: module number (0 - 7)

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

The procedure is used to check the identification code of PMS module 'mod_no'. It is a low level procedure that is called also during the initialisation in PMS_init. The procedure returns a module type value, if the id is correct. Possible module type values are defined in the pms_def.h file.

---

short CVICDECL **PMS_test_counter**(short mod_no, short *A_active, short *B_active);

Input parameters:

mod_no	module number (0 - 7)
* A_active	pointer to channel A gate polarity variable
* B_active	pointer to channel B gate polarity variable

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

The procedure performs a counters test in the PMS module 'mod_no' and determines the state of the gate polarity jumpers. The procedure tests at the beginning whether trigger works correctly. Possible values of A(B)_active variables after the test are listed below:

-1 - hardware error detected, gate polarity unknown

0 - test OK, gate polarity active high,

1 - test OK, gate polarity active low,

2 - hardware error detected. Probably jumper missing, check the module and insert the missing jumper

In case of a return value < 0 the function **PMS_get_test_error_string** can get additional information on test error. The error string is prepared during the test execution.

Important: The input signals must be disconnected from the module inputs during the test.

---

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

Input parameters:

error_string pointer to error message string

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

The procedure fills ‘error_string’ with the internal DLL string generated during the last execution of the **PMS_test_counter** or **PMS_init** function. ‘Error string’ contains detailed information on a counter test error or a DAC test error (during PMS_init).

After a call to **PMS_get_test_error_string** DLL’s internal error string is empty.

---

short CVICDECL **PMS_close**(void);

Input parameters: none

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

It is a low level procedure and not intended to normal use.

The procedure frees buffers allocated via DLL and set the DLL state as before PMS_init call.

PMS_init is the only procedure which can be called after PMS_close.

==========