SPECTRORADIOMETER
SOFTWARE DEVELOPMENT KIT

Release 3.1.01.0

Bentham Instruments Ltd
2 Boulton Road, Reading, Berkshire, RG2 0NH
Tel: +44 (0)118 975 1355, fax: +44 (0)118 931 2971,
email: sales@bentham.co.uk

Bentham Instruments Spectroradiometer Software Development Kit
Version 3.1.01.0

First printed January 1995, 1999 , this revision Jun 2011

CONTENTS

INTRODUCTION **1**

1. GETTING STARTED **3**

1.1. INDEX OF FILES ON THE DISK 3
1.2. INSTALLATION 4
1.3. MINIMUM SYSTEM REQUIREMENTS 4

2. THE HARDWARE CONTROL DLL **5**

2.1. WHAT IS A DLL 5
2.2. THE SYSTEM MODEL 5
2.2.1. List of Component types *8*
2.2.2. Component Types Alphabetical List *10*
2.3. COMPONENT GROUPS 15
2.4. DLL FUNCTIONS 17
2.5. HARDWARE ATTRIBUTE TOKENS 39
2.6. DLL ERROR CODES 44
2.7. CONTROLLING HARDWARE VIA THE DLL 45
2.7.1. M300/Mc300 and DM150/DMc150 Monochromators *45*
2.7.2. TM300/TMc300, DTM300/DTMc300 and TTM300 Monochromators *45*
2.7.3. Filter Wheel *46*
2.7.4. SAM *46*
2.7.5. TLS *47*
2.7.6. MVSS *47*
2.7.7. SOB *47*
2.7.8. 228, 228A, 485 and 487 ADCs *48*
2.7.9. 225, 265, 267, 277, 485,487 and 477 Amplifiers *48*
2.7.10. System Attributes *50*
2.8. USING THE DLL 51

3. TROUBLESHOOTING **53**

3.1. WINDOWS ERRORS 53
3.2. HARDWARE CONTROL PROBLEMS 53

INTRODUCTION

The Bentham Instruments Spectroradiometer Software Development Kit (SDK) is aimed at those who wish to develop their own applications to work with Bentham Instruments light measurement systems. The SDK centres on the use of the hardware control dynamic link library (DLL). The DLL provides a small but powerful suite of high-level functions that allow the developer to forget the low-level complexities of operating their hardware and concentrate instead on data acquisition and manipulation. The DLL is built from the code used in Bentham's own spectroradiometer control packages, meaning that it has been extensively tried and tested in a huge variety of systems.

This manual describes the use of the SDK. It is organised as follows:

Chapter 1 outlines the contents of the SDK and gives guidelines for installation.

Chapter 2 describes the spectrometer control DLL in detail, including services that it offers and how they are used.

Chapter 3 describes problems that you may encounter using the Spectroradiometer Control DLL and how to solve them.

1. GETTING STARTED

1.1. INDEX OF FILES ON THE DISK

\code examples **\C**
\Delphi
\Java
\LabView
\MATLAB
\Python
\VBA

Source code examples for the above languauges.

\hardware

system.cfg	-	system configuration file (where provided)
system.atr	-	system configuration file (where provided)

\lib

benhw32_fastcall.dll	-	32 bit hardware control DLL (Delphi _fastcall calling convention)
benhw32_stdcall.dll	-	32 bit hardware control DLL (Windows _stdcall calling convention)
benhw32_cdecl.dll	-	32 bit hardware control DLL (C/C++ _cdecl calling convention)
ieee_32m.dll	-	32 bit PC488 driver (4 versions included)

\lang\c

bendll.h	-	C header file
dllerror.h	-	C error code file
dlltoken.h	-	C attribute identifier file

\lang\generic

dllerror.txt	-	text error code file
dlltoken.txt	-	text attribute identifier file

\lang\pascal

bendll.pas	-	Pascal DLL import unit
dllerror.pas	-	Pascal error code file
dlltoken.pas	-	Pascal attribute identifier file

\lang\vb

bendll.bas	-	VB DLL function declaration module
dllerror.bas	-	VB error code file
dlltoken.bas	-	VB attribute identifier file

1.2. INSTALLATION

Win2000/WinXP/Win7 - copy the contents of \lib bit on the SDK disk to your system directory (typically c:\windows\system), or the directory where your application resides.

If your hardware is not PC488 controlled system then the ieee_32m.dll is a 'dummy' file but still needs to be installed. If you have a PC488 card then the copies and ieee_32m.dll will be the same as those on the disks accompanying your card; licensing information can be found in the PC488 manual.
Where possible a system configuration file, \hardware\system.cfg, written specifically for your hardware is included. This file is essential for using the spectroradiometer control DLL and may be copied as required. Unless new hardware is added, this file should never be edited.

To use the example programs you should have installed the appropriate libraries as described above. The example programs also expect the system configuration file system.cfg to be in the same directory as they are launched from.

1.3. MINIMUM SYSTEM REQUIREMENTS

Intel Pentium II / 300 MHz or better
Microsoft Windows 2000 Professional (SP4) or Windows XP Professional (SP2)
128Mb RAM or better
CD-ROM or DVD-ROM Drive
SVGA or higher-resolution monitor (XGA recommended)

2. THE HARDWARE CONTROL DLL

2.1. WHAT IS A DLL

A dynamic link library (DLL) is an executable module that contains code or resources that can be used by clients such as other applications or DLLs. When a client wishes to utilise code or resources in a DLL, the DLL is loaded into memory and linked to the client at runtime. The DLL does not have to be written in the same language as the client and can be easily replaced with updated versions when necessary. For this manual a DLL will be considered as a pre-compiled library of functions that can be called by other applications.

2.2. THE SYSTEM MODEL

In order for the DLL to be able to control a system it has to know what hardware it consists of. The DLL achieves this by constructing a **system model** that mirrors the hardware components. The system model is entirely modular, in the same way as the spectroradiometer system. It is the system model that co-ordinates the activities of the various pieces of hardware and allows almost any system, however large and complicated, to be controlled by the same set of simple, high-level functions.

The system model is built from a **system configuration** file. This describes the spectroradiometer in terms of what components it consists of, how the PC can communicate with them and how they interact. System configuration files rarely need to be changed, and where possible they are pre-written and supplied with the SDK.

Figure 1 shows a system configuration file that describes a simple TM300 based DC system.

```
#
#Demo TM300 system
#
USB_COMMS   comms       vid 1240, pid 5892
USB         msd         vid 1240, pid 5893
Amp487      dc_amp      address 64
ADC487      adc         address 72
FW252       fwheel      drive 2
SAM         exit_sam    drive 1
TM300       mono        use fwheel, use exit_sam
```

Figure 1: Example System Configuration File

Figure 1 describes the following system:
1 USB_COMMS - USB electronics controller at vid 1240 and pid 5892
1 USB – USB mechanical controller at vid 1240 and pid 5892
1 Amp487 - DC amplifier at address 64 on the I2C bus in the electronics bin
1 ADC487 - ADC at address 29 on the I2C bus in the electronics bin
1 252 filter wheel controlled by drive 2 of the MSD
1 SAM controlled by drive 1 of the MSD
1 TM300 monochromator housing the previously described SAM and filter wheel and MSD board

The format of the file is as follows:
- One component is described per line,
- For each line:
  - The first column declares the **type** of the component,
  - The second column gives the component an **identifier**,

- The third column contains any **parameters** (e.g. address), with multiple parameters separated by commas,
- Anything following a '#' on a line is a comment and is ignored.

The system configuration file is the means by which the DLL knows what hardware is present and how it is connected to the PC. It must contain one line for each PC controlled hardware module.
The first task when writing a system configuration is to declare how the PC communicates with any electronics in the system. Line 4 in figure 1 specifies that a USB_COMMS controller with vid 1240 and pid 5892 is to be used (other options include a PC488 IEEE controller, a TAS016 RS232<>IEEE converter, INES PCMCIA IEEE controller or an RS232, RS232 controller).
The next task is to declare how the PC communicates with any mechanical components in the system. Line 5 in figure 1 specifies that a USB controller with vid 1240 and pid 5893 is to be used (other options include a PMC, MSD or MAC which can be RS232 or IEEE).
These lines defining how the PC communicates must appear in the file before any other system component so the DLL knows how to communicate with the hardware in the system.

The next step is to add a line describing each hardware module in turn. A good 'bottom-up' approach is to start with each electronics module (e.g.amplifiers and ADC), then each mechanical/stepper motor drive controlled unit (e.g. filter wheel, SAMs, SOBs, MVSSs) and finally the monochromator.

One component is described per line. The first column is the **type**. This tells the DLL what the component is, e.g. 487 amplifier, 487 ADC, PMC. The next section gives a complete list of component types. The type is case sensitive.

The second column is an **identifier** by which the component can be referred to. The identifier can contain any alphanumeric characters and the '_' (underscore) character, but the first character must be a letter or '_'. Each identifier must be unique and cannot be any of the system configuration keywords (i.e. a type or parameter). The identifier is case sensitive.

The third column is a list of component **parameters**. These provide details such as the address of the component; the next section gives the valid parameters for each component type. If there is more than one parameter they must be separated by commas. The parameter list is case sensitive.

In systems with more than one electronics controller the entry for each device must specify which controller the component is connected to. This is achieved by the use parameter. Inserting the identifier of a controller as the value for the use parameter instructs the system model to operate the component via that controller. For example, the configuration file for a system with an USB_COMMS controlled 477Amp could contain:

```
USB_COMMS   comms2      vid 1240, pid 5891
477Amp      pre_amp     address 104, use comms2
```

In systems with more than one stepper motor drive (SMD) the entry for each SMD controlled component must specify which SMD the component is connected to. This in a similar way to the electronics. For example, the configuration file for a system with an MSC1 controlled MVSS could contain:

```
MSC1  slit_drive        address 20
MVSS  entrance_slit     motor 1, use slit_drive
```

This also specifies that the MVSS is to be addressed as motor 1. When there is more than one SMD in a system and the SMD is not specified for a component the DLL will assign a default, but it may not be the correct one.

The *use* parameter also allows some components (specifically filter wheels, SAMs and MVSSs) to be attached to the monochromator. The system model needs to know what components are part of the monochromator so that it can correctly co-ordinate all of their operations. This is illustrated in the final line of figure 1 which describes a TM300 that contains a SAM and filter wheel.

The *use* parameter is the reason for the suggested bottom-up order of adding components to the system configuration. This is because one component cannot be used by another unless it has been declared first.

Figure 2 is a more sophisticated example of a system configuration describing an AC system based on a DTM300 with a filter wheel, 2 SAMs and 3 MVSSs. The number of component types along with all of their different parameters may seem daunting, but once set up the system configuration file should never need altering. As the DLL functions access components via their identifiers as defined in the configuration file the only usual reason for looking at it is to find out what these identifiers are.

Once a system configuration file has been written it is passed to the DLL using the *BI_build_system_model* function. This instructs the DLL to compile a system model from the system configuration file. The function reports its success or failure and gives the reason for any failure to compile.

```
#-----------------------------------------------------------------
# Example system configuration file 2
#-----------------------------------------------------------------

# Specifies CEC488 controlled system
PC488      comms            address 21

# Defines an MSD3 connected to COM1 rather than the IEEE bus
MSD3       mono_drive       port COM1

# Defines a MAC at address 20
MAC        slit_drive       address 20, cards 3

# Defines a 277 pre-amplifier at address 28
Amp277     pre_amp          address 28

# Defines a 225 lock-in amplifier at address 27
Amp225     ac_amp           address 27

# Defines a 228A ADC at address 28
ADC228A    adc              address 29

# Defines 3 MVSSs, all controlled by the same MAC
MVSS       entrance_slit    drive 1, use slit_drive
MVSS       middle_slit      drive 2, use slit_drive
MVSS       exit_slit        drive 3, use slit_drive

# Defines a filter wheel controlled by card 3 of the MSD
FW252      fwheel           card 3, use mono_drive

# Defines a SAM controlled as SAM1 on card 1 of the MSD
SAM        exit1_sam        card 11, use mono_drive

# Defines a SAM controlled as SAM2 on card 1 of the MSD
SAM        exit2_sam        card 12, use mono_drive

# Defines a DTM300 containing all of the slits, SAMs and
# filter wheel and driven by the MSD.  Note that long lines are
# OK, but cannot contain any line breaks.
DTM300     mono             use fwheel, use exit1_sam, use exit2_sam,
use entrance_slit, use middle_slit, use exit_slit, use mono_drive
```

Figure 2: Example System Configuration File

2.2.1. List of Component types

This section gives a complete list of component types. Each type is described, along with its parameters.

Table 1: Complete List of Component Types

Component Type	Component Modelled	Parameters
PC488, CEC488	PC488 IEEE controller	address *0..30* port $*address*
TAS016, RS232	TAS016 RS232<>IEEE converter	port COM*n*
USB_COMMS	USB Interface	pid *n (5892)* vid *n (1240)*
MSD	MSD micro-stepping drive	address *0..30* port COM*n* cards *n*
MAC	MAC micro-stepping drive	address *0..30* port COM*n* cards *n*
USB	USB MAC micro-stepping drive	pid *n (5893)* vid *n (1240)*
MSC1 PMC	MSC1 micro-stepping drive PMC micr-stepping drive	address *0..30* address *0..30*
ADC228	228 ADC	address *0..30* use *identifier*
ADC228A	228A ADC	address *0..30* use *identifier*
ADC487	487 ADC	address *72,74,76,78* use *identifier*
ADC485	485 ADC	address *96,98,100,102* use *identifier*
Amp225	225 lock-in amplifier	address *0..30* use *identifier*
Amp265	265 DC amplifier	address *0..30* use *identifier*
Amp267	267 DC amplifier	address *0..30* use *identifier*
Amp277	277 pre-amplifier	address *0..30* use *identifier*
Amp487	487 DC amplifier	address *64,66,68,70* use *identifier*
Amp485	485 lock-in amplifier	address *88,90,92,94* use *identifier*
Amp477	477 pre-amplifier	address *104,106,108,110* use *identifier*
SB262 SB462	262 switch box 462 switch box	address *0..30* address 112..118
FW252	252 filter wheel	card *integer* positions *0..12* use *identifier*

IEEEDevice	Anonymous IEEE device	address *0..30* use *identifier*
USBDevice	Anonymous USB device	address 0..238 use *identifier*
Motor	Motorised stage	motor *1..3* use *identifier*
MVSS_MK1	MK1 Motorised Slit	resolution *n* drive 1..max Drives use *identifier*
MVSS_MK2	MK2 Motorised Slit	drive 1..max Drives use *identifier*
MVSS_MK3	MK3 Motorised Slit	drive 1..max Drives use *identifier*
EBox_Monitor	EBox Monitor	address 152 use *identifier*
SAM	Swing Away Mirror	card *integer* use *identifier*
SOB	Switch-over Box	card *integer* use *identifier*
TLS	Triple Light Source	card *integer* use *identifier*
M300E	M300E monochromator	use *identifier* selfpark *1* or *0*
M300HR	M300HR monochromator	as M300E
M300T	M300T monochromator	as M300E
DM150	DM150 monochromator	as M300E
TM300	TM300 monochromator	use *identifier* subtractive *1* or 0
DTM300	DTM300 monochromator	use *identifier* subtractive *1* or 0
TTM300	TTM300 monochromator	use *identifier*
HR600	HR600 monochromator	use *identifier* subtractive *1* or 0

2.2.2. Component Types Alphabetical List

The following is an alphabetical list of all of the component types that can be used in the system model, along with valid parameters and any other notes.

ADC228 *add a 228 ADC to the system model*

Parameters	address	the address of the 228
	use	specify which controller to use

ADC228A *add a 228A ADC to the system model*

Parameters	address	the address of the 228A
	use	specify which controller to use

ADC485 *add a 485 ADC to the system model*

Parameters	address	the address of the 485ADC
	use	specify which controller to use

ADC487 *add a 487 ADC to the system model*

Parameters	address	the address of the 487ADC
	use	specify which controller to use

Amp225 *add a 225 AC amplifier to the system model*

Parameters	address	the address of the amplifier
	use	specify which controller to use

Amp265 *add a 265 DC amplifier to the system model*

Parameters	address	the address of the amplifier
	use	specify which controller to use

Amp267 *add a 267 DC amplifier to the system model*

Parameters	address	the address of the amplifier
	use	specify which controller to use

Amp277 *add a 277 pre-amplifier to the system model*

Parameters	address	the address of the amplifier
	use	specify which controller to use

Amp477 *add a 477 pre-amplifier to the system model*

Parameters	address	the address of the amplifier
	use	specify which controller to use

Amp485 *add a 485 AC amplifier to the system mod***el**

Parameters	address	the address of the amplifier
	use	specify which controller to use

Amp487 *add a 487 DC amplifier to the system model*

Parameters	address	the address of the amplifier
	use	specify which controller to use

CEC488 *add a CEC488 (or compatible) interface card*

Parameters	address	the IEEE address of the card
	port	the memory address of the card (only required if card not at default address, e.g. INES PCMCIA IEEE card)

DM150 *add a DM150 monochromator to the system model*

Parameters	use	add a slave component to the DM150 or specify which SMD to use
	selfpark	a value of 1 indicates self-park capability

Notes Valid slave components are filter wheels (5), SAMs (10) and MVSSs (5). The selfpark parameter is only used if the DM150 is PMC or MSC1 controlled.

DTM300 *add a DTM300 monochromator to the system model*

Parameters	use	adds a slave component to the DTM300 or specify which SMD to use

Notes Valid slave components are filters wheels (5), SAMs (10) and MVSSs (5).

EBox_Monitor *add an ebox monitor to the system model*

Parameters	address	the address of the ebox monitor
	use	specify which controller to use

FW252 *define a 252 filter wheel for use in a monochromator*

Parameters	use	the MAC, MSD or PMC that the filter wheel is connected to
	card	which MSD card the filter wheel is connected to
	drive	which MAC drive the filter wheel is connected to
	positions	the number of filter positions (default is 6)
	selfpark	a value of 1 indicates self-park capability

Notes The selfpark parameter is only used if it is PMC or MSC1 controlled. The positions parameter is optional; the default (6) is used if omitted.

HR600 *add a HR600 monochromator to the system model*

Parameters	use	add a slave components to the HR600 or specify which SMD to use

Notes Valid slave components are filters wheels (5), SAMs (10) and MVSSs (5).

IEEEDevice *add an anonymous IEEE Device*

Parameters	address	the address of the device
	use	specify which controller to use

M300E *add an M300E monochromator to the system model*

Parameters	use	add a slave component to the M300E or specify which SMD to use
	selfpark	a value of 1 indicates self-park capability

Notes Valid slave components are filter wheels (5), SAMs (10) and MVSSs(5). The selfpark parameter is only used if the M300E is PMC or MSC1 controlled.

M300HR *add an M300HR monochromator to the system model*

Parameters	use	add a slave component to the M300HR or specify which SMD to use
	selfpark	a value of 1 indicates self-park capability

Notes Valid slave components are filter wheels (5), SAMs (10) and MVSSs(5).

The selfpark parameter is only used if the M300HR is PMC or MSC1 controlled.

M300T *add an M300T monochromator to the system model*

Parameters	use	add a slave component to the M300T or specify which SMD to use
	selfpark	a value of 1 indicates self-park capability

Notes Valid slave components are filter wheels (5), SAMs (10) and MVSSs(5). The selfpark parameter is only used if the M300T is PMC or MSC1 controlled.

MAC *add a MAC to the system model*

Parameters	address	address of the MAC
	cards	number of motors the MAC can control

MOTOR *add a motorised stage to the system model*

Parameters	use	which SMD the motor is driven by
	drive	which motor/drive the SMD refers to this as
	resolution	the number of steps per unit (e.g. steps/mm, steps/°)

MSC1 *add an MSC1 to the system model*

Parameters	address	address of the MSC1

MSD *add an MSD to the system model*

Parameters	address	address of the MSD (when IEEE controlled)
	port	serial port the MSD is connected to (when under direct PC RS232 control)
	cards	number of controller cards in the MSD

Notes The address and port parameters are mutually exclusive; the MSD is either IEEE or RS232 controlled.

MVSS-MK1..Mk3 *define a motorised slit for use in a monochromator*

Parameters	use	-	SMD which the slit is connected to
	motor	-	which motor/drive of the SMD the MVSS is connected to

Notes A motor drive must be specified with the use parameter. For a PMC controlled MVSS the motor parameter can be 1 or 2. For an MSC1 controlled MVSS it can be 1, 2 or 3. For MAC and IMAC controllers the motor parameter can be any of the available drives up to the max number of drives.

PC488 *add a PC488 (or compatible) interface card*

Parameters	address	the IEEE address of the card
	port	the memory address of the card (only required if card not at default address, e.g. INES PCMCIA IEEE card)

PMC *adds a PMC to the system model*

Parameters	address	address of the PMC

RS232 *communicate with hardware via RS232*

Parameters	port	serial port to be used (e.g. COM1)

SAM *add a SAM to the system model or define it for use in a monochromator*

Parameters	use	MAC, IMAC, MSD or PMC that the SAM is connected to
	card	which card the SAM is connected to (MSD control)
	drive	which drive the SAM is connected to (MAC control)

Notes For MACs, and MSDs that can control two SAMs or SOBs per card, the drive/card parameter specifies the drive/card number and SAM number. For example, drive 21 refers to SAM1 on drive 2 of a MAC; the first digit is the card, the second the SAM. For IMACs and MSDs that are only capable of operating one SAM/SOB per card, the card parameter is a single digit specifying the card number.

SB262 *add a 262 to the system model*

Parameters	address-	address of the 262
	use	specify which controller to use

SB462 *add a 462 to the system model*

Parameters	address	address of the 462
	use	specify which controller to use

SOB *add a SOB to the system model or define it for use in a monochromator*

Parameters	use	MAC, IMAC, MSD or PMC that controls the SOB
	card	which card controls the SOB (MSD)
	drive	which drive controls the SOB (MAC)

Notes For MACs (or MSDs that can control two SAMs or SOBs per card) the drive (card) parameter specifies the drive (card) number and SOB number. For example, drive 21 refers to SOB1 on drive 2 of a MAC; the first digit is the drive, the second the SOB. For MSDs that are only capable of operating one SAM/SOB per card, the card parameter is a single digit specifying the card number.

TAS016 *communicate with hardware via a TAS016*

Parameters	port	serial port that the TAS016 is connected to (e.g. COM1)

TLS *add a Triple Light source to the system model*

Parameters	use	SMD that the SAM is connected to
	card	which card the SAM is connected to (MSD control)
	drive	which drive the SAM is connected to (MAC control)

TM300 *add a TM300 monochromator to the system model*

Parameters	use	add a slave components to the TM300 or specify which SMD to use

Notes Valid slave components are filters wheels (5), SAMs (10) and MVSSs (5).

TTM300 *add a TTM300 monochromator to the system model*

Parameters	use	add a slave components to the TTM300 or specify which SMD to use

Notes Valid slave components are filters wheels (5), SAMs (10) and MVSSs (5).

USB *add USB mechanical controller to the system model*

Parameters	pid	the product id of the USB hid device (default 5893)
	vid	the vendor id of the USB hid device (default 1240)

USBDevice *add an anonymous USB device to the system model*

Parameters	address	the address of the device
	use	specify which controller to use

USB_COMMS *add USB electronics controller to the system model*

Parameters	pid	the product id of the USB hid device (default 5892)
	vid	the vendor id of the USB hid device (default 1240)

2.3. COMPONENT GROUPS

Simple spectroradiometer systems contain the same basic set of components: a monochromator, a stepper motor drive, an amplifier, possibly a pre-amplifier and an ADC. Measurements are made by repeatedly selecting a wavelength and taking a reading. In this situation all of the components are used together.

More sophisticated systems are possible. For example, consider a system designed to take both AC and DC measurements, possibly during the same scan. This time measurements would be made by selecting a wavelength and taking a reading using just one of the sets of detection electronics. In this situation the system model needs to know what set of components it should be using to perform the required operation.

To allow this the system model includes the concept of **component groups**. A component group is a set of components that are used together to perform operations. At any given time there is one **active group**; this is the component group that is used for all operations, such as selecting a wavelength or taking a measurement. A component group is a sub-set of the system model.

New component groups are added with the *BI_build_group* function. This constructs a new, empty component group and returns its sequence number. The first group is group 1, the next group created is group 2, the next group 3, and so on. There can be up to 10 component groups. When the system model is built, the DLL constructs a single empty default component group (group 1).

Existing component groups can be edited by using the *BI_group_add* and *BI_group_remove* functions to add and remove components to and from groups. For each function the component is referred to by its identifier as set in the system configuration file and the group by its sequence number.

The function *BI_use_group* sets the active group. All of the hardware operation functions (*BI_initialise*, *BI_close_shutter*, *BI_park*, *BI_zero_calibration*, *BI_select_wavelength*, *BI_autorange* and *BI_measurement*) work with the active group. Obviously for simple systems the active group will always be the default group (group 1).

```
#-------------------------------------------------
# Example system configuration
#-------------------------------------------------

PC488        comms             address 21
MAC          smd               address 30, cards 2

# Note DC and AC detection electronics
Amp267       dc_amp            address 26

Amp277       pre_amp           address 28
Amp225       ac_amp            address 25

ADC228A      adc               address 29

# SOB to select DC or AC input to ADC
SOB          adc_input         drive 11
FW252        fwheel            drive 2

TM300        mono              use fwheel
```

Figure 3: Example System Configuration Containing AC and DC Detectors

Figure 3 shows a system configuration for a TM300 based system with both AC and DC detection electronics. The input to the ADC (from the 267 or 225) is selected by the SOB,

which is controlled by the MAC. In order to make AC and DC measurements two groups, one containing the monochromator, MSD, SOB, ADC and 277 and 225, and one containing the monochromator, MSD, SOB, ADC and 267, must be created. This can be achieved using the following sequence of DLL function calls:

```
i = BI_build_group()
BI_group_add( "mono", i )
BI_group_add( "smd", i )
BI_group_add( "adc_input", i )
BI_group_add( "adc", i )
BI_group_add( "pre_amp", i )
BI_group_add( "ac_amp", i )

i = BI_build_group()
BI_group_add( "mono", i )
BI_group_add( "smd", i )
BI_group_add( "adc_input", i )
BI_group_add( "adc", i )
BI_group_add( "dc_amp", i )
```

Assuming that when the SOB is relaxed/off the ADC is reading from the 225, the following sequence of commands is a very naive example of a simple measurement scan:

```
for wl = start_wl to stop_wl by increment
   begin
      {Measurement with group 1 (AC)}
      BI_use_group( 1 )
      BI_set( "adc_input", SOBState, 0, 0 )
      BI_select_wavelength( wl, settle_time )
      wait( settle_time )
      BI_autorange()
      BI_measurement( reading )

      {Measurement with group 2 (DC)}
      BI_use_group( 2 )
      BI_set( "adc_input", SOBState, 0, 1 )
      BI_select_wavelength( wl, settle_time )
      wait( settle_time )
      BI_autorange()
      BI_measurement( reading )
   end
```

BI_zero_calibration needs to access the monochromator in order to correctly measure the zero offset and dark current for each state that the system will be in over the specified wavelength range. *BI_measurement* also needs to access the monochromator in order to retrieve the zero calibration values for the current wavelength. This means that if measurements performed using a group are to be zero calibrated the group must include the monochromator.

While the DLL will not create default component groups, *BI_save_setup* records which components are in which groups in the system attributes file and *BI_load_setup* deletes any existing component groups and builds new ones from the information in the system attribute file.

2.4. DLL FUNCTIONS

This section gives a complete description of each function in the spectroradiometer control DLL. The functions are listed in alphabetical order. C, Pascal and Visual Basic prototypes are given for each function. For those using different languages, the parameter types are as follows:

C	Pascal	Visual Basic	Description
int	integer	ByVal .. As Integer	16 bit signed integer
unsigned int	word	ByVal .. As Integer	16 bit unsigned integer
double	double	ByVal .. As Double	64 bit real number
char far *	Pchar	ByVal .. As String	pointer to a null terminated string
int far *	var int	As Integer	pointer to a 16 bit signed integer
unsigned int far *	var word	As Integer	pointer to a 16 bit unsigned value
long far *	var longint	As Long	pointer to a 32 bit signed integer

Table 1: DLL Function Parameter Types

For Visual Basic the symbol ↳ has been used as a continuation character. This indicates that a line of code has been split in the text but should be entered on one line in Visual Basic.

BI_automeasure

Syntax BI_automeasure(reading)

Description This function auto-ranges the amplifier(s) in the active group and returns the reading at the current wavelength. It takes the ADC offset and dark current (previously obtained by BI_zero_calibration) into account

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters reading - set to the reading at the current wavelength (if call is successful)

C int far pascal BI_automeasure(double far *reading);

Pascal function BI_ automeasure (var reading : double) : integer;

Visual Basic Function BI_ automeasure (reading As Double) As Integer

BI_autorange

Syntax BI_autorange

Description This function auto-ranges the amplifier(s) in the active group.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters None

C int far pascal BI_autorange(void);

Pascal function BI_autorange : integer;

Visual Basic Function BI_autorange( ) As Integer

BI_build_group

Syntax BI_build_group

Description This function constructs a new, empty component group. The DLL allows up to 10 component groups.

Return value If the function was successful it returns the group number, otherwise it returns BI_error.

Parameters None

C int far pascal BI_build_group(void);

Pascal function BI_build_group: integer;

Visual Basic Function BI_build_group( ) As Integer

BI_build_system_model

Syntax BI_build_system_model(filename, error_report)

Description This function instructs the DLL to build a system model from a specified system configuration file. This must be done before the DLL can perform any hardware control functions. The SDK is supplied with a customised system configuration file (system.cfg).

This function *must* succeed before any other DLL function is called. If there is any error the function reports it via *BI_report_error*.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters filename - the path name of the system configuration file
error_report - contains error report if function failed (256 characters maximum)

C
```
int far pascal BI_build_system_model(char far *filename,
                                     char far  *error_report );
```

Pascal
```
function BI_build_system_model(filename: Pchar;
                               error_report  : Pchar ) : integer;
```

Visual Basic
```
Function BI_build_system_model( ByVal filename As String,
                                ↳ByVal error_report As String ) As Integer
```

Notes In Visual Basic error_report must be declared as a fixed-length string, ie

```
Dim error_report As String * 256
```

BI_camera_measurement

Syntax BI_camera_measurement(id, wls, readings)

Description This function instructs the DLL to take a camera measurement using the camera defined by the id string. Two arrays are passed in, the first of which will be filled with the camera wavelengths, and the second the intensity readings. The camera measurement itself is performed by the external dll defined in the system configuration file for the relevant camera.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters id - component identifier as set in the system configuration file
wls - pointer to an array of doubles
readings - pointer to an array of doubles

C
```
int far pascal BI_camera_measurement (char far *id
                                      double far *wls
                                      double far *reading);
```

Pascal
```
function BI_ camera_measurement (id: pchar; wls: pdouble; readings:
pdouble ): integer;
```

Visual Basic
```
Function BI_ camera_measurement (ByVal id As String, wls As Double,
readings As Double) As Integer
```

BI_close

Syntax BI_close

Description This function instructs the DLL to destroy the system model and prepare for unloading.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters None

C int far pascal BI_close(void);

Pascal function BI_close: integer;

Visual Basic Function BI_close ( ) As Integer

BI_close_shutter

Syntax BI_close_shutter

Description This function instructs the DLL to send the filter wheel in the monochromator of the active group to its shutter position.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters None

C int far pascal BI_close_shutter(void);

Pascal function BI_close_shutter : integer;

Visual Basic Function BI_close_shutter( ) As Integer

BI_component_select_wl

Syntax BI_component_select_wl(id, wavelength, delay)

Description This function sends the specified component to the specified wavelength and recommends a settle delay time before any readings are taken. It does not perform the delay itself.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters
id - component identifier as set in the system configuration file
wavelength - wavelength to go to (nm)
delay - recommended delay (ms) before taking readings

C
```
int far pascal BI_component_select_wl (char far *id,double wavelength,
                                       long far*delay );
```

Pascal
```
function BI_component_select_wl (id: Pchar; wavelength: double;
                                  var delay: longint ) : integer;
```

Visual Basic
```
Function BI_component_select_wl (ByVal id As String,
    ↳ByVal wavelength As Double, delay As Long ) As Integer
```

BI_delete_group

Syntax BI_delete_group(n)

Description This function deletes the specified group from the system model

Return value The return value indicates either the result of the call:
BI_error - failure
Or the number of groups remaining after the deletion.

Parameters n - group number to be deleted

C
```
int far pascal BI_delete_group(int token)
```

Pascal
```
function BI_delete_group ( n : integer; ) : integer;
```

Visual Basic
```
Function BI_delete_group (ByVal n As Integer ) As Integer
```

BI_get

Syntax BI_get(id, token, index, value)

Description This function returns the value of the specified component attribute.

Return value The return value indicates the result of the call:
BI_OK - success
BI_invalid_token - invalid token
BI_invalid_component - the component does not exist in this system
BI_invalid_attribute - the attribute is not accessible in this system

Parameters
id - component identifier as set in the system configuration file
token - the token for attribute to be retrieved
index - for accessing indexed attributes and setups
value - set to the value of the attribute (if call successful)

C
```
int far pascal BI_get(          char far *id
                        int     token,
                        int     index
                        double far *value );
```

Pascal
```
function BI_get(    id      : Pchar;
                    token   : integer;
                    index   : integer;
                var value   : double ) : integer;
```

Visual Basic
```
Function BI_get(        ByVal id As String, ByVal token As Integer,
                ↳ByVal index As Integer, value As Double ) As Integer
```

BI_get_c_group

Syntax BI_get_c_group (group)

Description This function returns the current group number

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters group - set to the current group index (if call is successful)

C
```
int far pascal BI_get_c_group ( int far * group);
```

Pascal
```
function BI_get_c_group ( var group: integer ) : integer;
```

Visual Basic
```
Function BI_get_c_group (group As integer ) As Integer
```

BI_get_component_list

Syntax BI_get_component_list (list)

Description This function returns a list of all components in the system model.

NOTE : this does not return components contained within monochromators, to retrieve this information run BI_get_mono_items on all monochromators in the system.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters list - a comma-delimited list of the identifiers of the components in the system model

C
```
int far pascal BI_get_component_list (char far *list);
```

Pascal
```
function BI_get_component_list (list:pchar) : integer;
```

Visual Basic
```
Function BI_get_component_list (ByVal list As String) As Integer
```

Notes In Visual Basic description must be declared as a fixed-length string

BI_get_group

Syntax BI_get_group(group, description)

Description This function returns a list of the identifiers of all of the components in the specified group.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters group - the number of the group
description - a comma-delimited list of the identifiers of the components in the group

C
```
int far pascal BI_get_group( int          group,
                             char far  *description );
```

Pascal
```
function BI_get_group(  group       : integer;
                        description  : Pchar ) : integer;
```

Visual Basic
```
Function BI_get_group( ByVal group As Integer, ByVal description As String
                       ↳) As Integer
```

Notes In Visual Basic description must be declared as a fixed-length string

BI_get_hardware_type

Syntax BI_get_hardware_type (id, hardware_type)

Description This function returns the hardware type of the specified component

NOTE : The hardware_type codes can be found in the dlltokens file.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters id: component identifier as set in the system configuration file
hardware_type: an integer token indicating the hardware type

C int far pascal BI_get_hardware_type(char far*id, int far* hardware_type);

Pascal function BI_get_hardware_type(id: pchar; hardware_type : pinteger): integer;

Visual Basic Function BI_get_hardware_type(ByVal id As String,
ByVal hardware_type As Integer) As Integer

BI_get_max_bw

Syntax BI_get_max_bw (group,start_wl,stop_wl,bandwidth)

Description This function returns the maximum bandwidth that the slits may be set to across the given wavelength range

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters group - group number currently in use
start_wl - the start wavelength
stop_wl - the stop wavelength
bandwidth - the return value of maximum bandwidth across the wavelength range

C int far pascal BI_get_max_bw(int group, double start_wl, double stop_wl, double far* bandwidth);

Pascal BI_get_max_bw(group: integer; start_wl,stop_wl : double; var bandwidth : double):integer;

Visual Basic Function BI_get_max_bw(ByVal group As Integer,
ByVal start_wl As Double,
ByVal stop_wl As Double,
ByVal bandwidth As Double) As Integer

BI_get_min_step

Syntax BI_get_min_step (group,start_wl,stop_wl,min_step)

Description This function returns the minimum step that the monochromator is capable of, given the wavelength range and grating used

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters group - group number currently in use
start_wl - the start wavelength
stop_wl - the stop wavelength
min_step - the return value of min_step across the wavelength range (in nm)

C int far pascal BI_get_min_step(int group, double start_wl, double stop_wl, double far* min_step);

Pascal BI_get_min_step(group: integer; start_wl,stop_wl : double; var min_step : double):integer;

Visual Basic Function BI_get_min_step(ByVal group As Integer,
ByVal start_wl As Double,
ByVal stop_wl As Double,
ByVal min_step As Double) As Integer

BI_get_mono_items

Syntax BI_get_mono_items (id, list)

Description This function returns a list of all components contained within the specified monochromator

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters id: component identifier as set in the system configuration file
list - a comma-delimited list of the identifiers of the components in the monochromator

C int far pascal BI_get_mono_items(char far*id char far*list);

Pascal function BI_get_mono_items (id: pchar; list: pchar): integer;

Visual Basic Function BI_get_mono_items(ByVal id As String,
ByVal list As String) As Integer

Notes In Visual Basic description must be declared as a fixed-length string

BI_get_n_groups

Syntax BI_get_n_groups(n)

Description This function returns the number of groups in the system model

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters n - number of groups

C int far pascal BI_get_n_groups(int far* n);

Pascal function BI_get_n_groups(var n: integer) : integer;

Visual Basic Function BI_get_n_groups(ByVal n As Integer) As Integer

BI_get_no_of_dark_currents

Syntax BI_get_no_of_dark_currents (num)

Description This function returns the number of dark currents taken during the last zero calibrate

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters num: the number of dark currents taken during the last zero calibrate for the current group

C int far pascal BI_get_no_of_dark_currents (int far* num);

Pascal function BI_get_no_of_dark_currents (var num: integer): integer;

Visual Basic Function BI_get_no_of_dark_currents (ByVal num As Integer) As Integer

BI_get_str

Syntax BI_get_str(id, token, index, s)

Description This function returns the value of the specified component attribute, where the value is a string

Return value The return value indicates the result of the call:
BI_OK - success
BI_invalid_token - invalid token
BI_invalid_component - the component does not exist in this system
BI_invalid_attribute - the attribute is not accessible in this system

Parameters
id - component identifier as set in the system configuration file
token - the token for attribute to be retrieved
index - for accessing indexed attributes and setups
s - set to the string of the attribute (if call successful)

C
```
int far pascal BI_get(      char far *id
                    int     token,
                    int     index
                    char far *s );
```

Pascal
```
function BI_get(    id    : Pchar;
                    token : integer;
                    index : integer;
                    s     : Pchar ) : integer;
```

Visual Basic
```
Function BI_get(    ByVal id As String, ByVal token As Integer,
                    ByVal index As Integer, n As String ) As Integer
```

BI_get_zero_calibration_info

Syntax BI_get_zero_calibration_info (wavelength,darkcurrent,adc_offset)

Description This function returns lists of wavelengths, dark currents and adc_offsets of all points from the last zero calibrate on the current group.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters wavelength: pointer to an array of doubles that the function fills with the wavelengths of zero calibrations
darkcurrent: pointer to an array of doubles that the function fills with the dark currents from zero calibration points
adc_offset: pointer to an array of doubles that the function fills with the adc offsets from zero calibration points

C
```
int far pascal BI_get_zero_calibration_info (double far*wavelength,
                                             double far* darkcurrent,
                                             double far* adc_offset);
```

Pascal
```
function BI_get_zero_calibration_info (wavelength: pdouble
                                       darkcurrent: pdouble
                                       adc_offset: pdouble): integer;
```

Visual Basic
```
Function BI_get_zero_calibration_info (ByVal wavelength As Double,
    ↳ByVal darkcurrent As Double
    ↳ByVal adc_offset As Double) As Integer
```

BI_group_add

Syntax BI_group_add(id, group)

Description This function adds a component to the specified group. If the group already contains a component of the same type it is replaced by the new one.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters id - component identifier as set in the system configuration file
group - number of the group to add the component to

C
```
int far pascal BI_group_add(char far*id, integer group );
```

Pascal
```
function BI_group_add(id: Pchar; group: integer ) : integer;
```

Visual Basic
```
Function BI_group_add( ByVal id As String, ByVal group As Integer ) As Integer
```

BI_group_remove

Syntax BI_group_remove(id, group)

Description This function removes a component from the specified group.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters id - component identifier as set in the system configuration file
group - number of the group to remove the component from

C int far pascal BI_group_remove(char far*id, int group);

Pascal function BI_group_remove(id: Pchar; group: integer) : integer;

Visual Basic Function BI_group_remove(ByVal id As String, ByVal group As Integer)
↳As Integer

BI_initialise

Syntax BI_initialise

Description This function initialises the active group as follows:

MSD - set-up to operate monochromator
SAMs - sent to initial position
265 - set to start range
267 - set to start gain and channel for the current set-up
277 - set to start gain and channel for the current set-up
225 - set to start gain, channel, phase variable and quadrant, frequency mode and time constant for the current set-up

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters None

C int far pascal BI_initialise(void);

Pascal function BI_initialise( ): integer;

Visual Basic Function BI_initialise( ) As Integer

NOTE : This function *must* be successfully called before the DLL performs any other hardware control functions.

BI_load_setup

Syntax BI_load_setup(filename)

Description This function sets up the DLL system model from a specified system attribute file (created using BI_save_setup). The SDK is supplied with an example system attribute file (system.atr).

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters filename - the path name of the system attribute file

C int far pascal BI_load_setup(char far *filename);

Pascal function BI_load_setup(filename : Pchar) : integer;

Visual Basic Function BI_load_setup(ByVal filename As String) As Integer

BI_measurement

Syntax BI_measurement(reading)

Description Using the active group, this function returns the reading at the current wavelength. It takes the ADC offset and dark current (previously obtained by BI_zero_calibration) into account.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters reading - set to the reading at the current wavelength (if call is successful)

C int far pascal BI_measurement(double far *reading);

Pascal function BI_measurement(var reading : double) : integer;

Visual Basic Function BI_measurement(reading As Double) As Integer

BI_multi_automeasure

Syntax BI_multi_automeasure(reading)

Description This function auto-ranges the amplifier(s) in each group and returns an array of readings at the current wavelength (one for each group). It takes the ADC offset and dark current (previously obtained by BI_multi_zero_calibration) into account

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters reading - set to the reading of the first group at the current wavelength (if call is successful). The reading for subsequent groups is retrieved by incrementing the pointer.

C int far pascal BI_automeasure(double far* reading);

Pascal function BI_ automeasure (reading : pdouble) : integer;

Visual Basic Function BI_ automeasure (reading As Double) As Integer

BI_multi_autorange

Syntax BI_multi_autorange

Description This function auto-ranges the amplifier(s) in all groups in the system.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters None

C int far pascal BI_multi_autorange (void);

Pascal function BI_multi_autorange: integer;

Visual Basic Function BI_multi_autorange ( ) As Integer

BI_multi_get_no_of_dark_currents

Syntax BI_multi_get_no_of_dark_currents (group, num)

Description This function returns the number of dark currents taken during the last zero calibrate

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters group- number of the group to get the number of dark currents from
num: the number of dark currents taken during the last zero calibrate for the specified group

C int far pascal BI_multi_get_no_of_dark_currents (int group,int far* num);

Pascal function BI_multi_get_no_of_dark_currents (group : integer;
var num: integer): integer;

Visual Basic Function BI_multi_get_no_of_dark_currents (ByVal group As Integer
↳ByVal num As Integer) As Integer

BI_multi_get_zero_calibration_info

Syntax BI_multi_get_zero_calibration_info (group, wavelength, darkcurrent,
adc_offset)

Description This function returns lists of wavelengths, dark currents and adc_offsets of all points from the last zero calibrate on the specified group.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters group- number of the group to get the zero calibration info from
wavelength: pointer to an array of doubles that the function fills with the wavelengths of zero calibrations
darkcurrent: pointer to an array of doubles that the function fills with the dark currents from zero calibration points
adc_offset: pointer to an array of doubles that the function fills with the adc offsets from zero calibration points

C int far pascal BI_multi_get_zero_calibration_info (int: group,
double far*wavelength,
double far* darkcurrent,
double far* adc_offset);

Pascal function BI_multi_get_zero_calibration_info (group : integer;
wavelength: pdouble
darkcurrent: pdouble
adc_offset: pdouble): integer;

Visual Basic Function BI_multi_get_zero_calibration_info (ByVal group As Integer
↳ByVal wavelength As Double,
↳ByVal darkcurrent As Double
↳ByVal adc_offset As Double) As Integer

BI_multi_initialise

Syntax BI_multi_initialise

Description This function initialises all group as follows:

MSD - set-up to operate monochromator
SAMs - sent to initial position
265 - set to start range
267 - set to start gain and channel for the current set-up
277 - set to start gain and channel for the current set-up
225 - set to start gain, channel, phase variable and quadrant, frequency mode and time constant for the current set-up

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters None

C int far pascal BI_multi_initialise(void);

Pascal function BI_multi_initialise( ): integer;

Visual Basic Function BI_multi_initialise( ) As Integer

NOTE : This function must be successfully called before the DLL performs any other multi group hardware control functions.

BI_multi_measurement

Syntax BI_multi_measurement (reading)

Description This function returns the readings at the current wavelength for all groups. It takes the ADC offset and dark current (previously obtained by BI_multi_zero_calibration) into account.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters reading – pointer to an array of doubles in which the function stores the readings at the current wavelength (if call is successful)

C int far pascal BI_multi_measurement (double far *reading);

Pascal function BI_multi_measurement (reading : pdouble) : integer;

Visual Basic Function BI_multi_measurement (reading As Double) As Integer

BI_multi_park

Syntax BI_multi_park

Description This function parks the monochromator of all the groups if possible (e.g. a TM300, DTM300 or M300/DM150 capable of self-parking) and leaves the DLL and monochromator in a well-defined state. Where a monochromator can be parked this function must be called before BI_zero_calibration and BI_select_wavelength are used. BI_multi_park only needs to be called once; after this the DLL records the state of the monochromator(s).

Parking a monochromator parks all turrets, the filter wheel and all MVSSs.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters None

C
```
int far pascal BI_multi_park( void );
```

Pascal
```
function BI_multi_park : integer;
```

Visual Basic
```
Function BI_multi_park( ) As Integer
```

BI_multi_select_wavelength

Syntax BI_multi_select_wavelength (wavelength, delay)

Description This function performs the following operations with all groups in the system:

- sends the monochromator to the specified wavelength,
- selects the filter for the specified wavelength,
- positions all SAMs according to the specified wavelength,
- sets all MVSSs according to the specified wavelength
- configures all amplifiers for the specified wavelength

and recommends a settle delay time before any readings are taken. It does not perform the delay itself.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters wavelength - wavelength to go to (nm)
delay - recommended delay (ms) before taking readings

C
```
int far pascal BI_multi_select_wavelength(      double     wavelength,
                                                long far   *delay );
```

Pascal
```
function BI_multi_select_wavelength(            wavelength      : double;
                                        var delay               : longint ) : integer;
```

Visual Basic
```
Function BI_multi_select_wavelength( ByVal wavelength As Double,
                                        ↳delay As Long ) As Integer
```

BI_multi_zero_calibration

Syntax BI_multi_zero_calibration (start_wavelength, stop_wavelength)

Description This function performs ADC offset and system dark current measurements for all group. The DLL records the values and uses them when calculating readings in BI_multi_measurement. To ensure accurate readings BI_multi_zero_calibration should be called before BI_multi_measurement is used; once before a wavelength range scan is normally sufficient.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters start_wavelength – wavelength to zero calibrate from.
stop_wavelength – wavelength to zero calibrate to.

C
```
int far pascal BI_multi_zero_calibration (double start_wavelength,
                                          double stop_wavelength );
```

Pascal
```
function BI_multi_zero_calibration (start_wavelength: double;
                                    stop_wavelength: double ) : integer;
```

Visual Basic
```
Function BI_multi_zero_calibration ( ByVal start_wavelength As Double,
                                     ByVal stop_wavelength As Double ) As Integer
```

BI_park

Syntax BI_park

Description This function parks the monochromator of the active group if possible (e.g. a TM300, DTM300 or M300/DM150 capable of self-parking) and leaves the DLL and monochromator in a well-defined state. Where a monochromator can be parked this function must be called before BI_zero_calibration and BI_select_wavelength are used. BI_park only needs to be called once; after this the DLL records the state of the monochromator.

Parking the monochromator parks all turrets, the filter wheel and all MVSSs.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters None

C
```
int far pascal BI_park( void );
```

Pascal
```
function BI_park : integer;
```

Visual Basic
```
Function BI_park( ) As Integer
```

BI_read

Syntax BI_read(message, buffer_size, chars_read, id)

Description This function reads message from the device at the specified IEEE address. It is intended to be used for communicating with devices that are not controlled by any other DLL functions (e.g. non-standard or third party hardware).

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters
message - a pointer to a buffer for storing any message from the device
buffer_size - the maximum number of characters to read
chars_read - the number of characters actually read
id - component identifier as set in the system configuration file

C
```
int far pascal BI_read(  char far   *message,
                         word       buffer_size,
                         word far   *chars_read,
                         char far*  id );
```

Pascal
```
function BI_read(        message    : pchar;
                         buffer_size: word;
                     var chars_read : word;
                         id   : pchar) : integer;
```

Visual Basic
```
Function BI_read( ByVal message As String, ByVal buffer_size As Integer,
                  ↳chars_read As Integer, ByVal id As String ) As Integer
```

BI_report_error

Syntax BI_report_error

Description This function returns an error code corresponding to the last hardware error encountered. Calling this function resets the error code in the DLL.

Return value Indicates the last hardware error encountered.

Parameters None

C
```
int far pascal BI_report_error( void );
```

Pascal
```
function BI_report_error : integer;
```

Visual Basic
```
Function BI_report_error( ) As Integer
```

BI_save_setup

Syntax BI_save_setup(filename)

Description This function creates a system attribute file for the current state of the DLL system model. This file can be reloaded with BI_load_setup.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters filename - the path name of the set up file

C
```
int far pascal BI_save_setup( char far *filename );
```

Pascal
```
function BI_save_setup( filename : Pchar ) : integer;
```

Visual Basic
```
Function BI_save_setup( ByVal filename As String ) As Integer
```

BI_select_wavelength

Syntax BI_select_wavelength(wavelength, delay)

Description Using the active group, this function performs the following operations:

- sends the monochromator to the specified wavelength,
- selects the filter for the specified wavelength,
- positions all SAMs according to the specified wavelength,
- sets all MVSSs according to the specified wavelength
- configures all amplifiers for the specified wavelength

and recommends a settle delay time before any readings are taken. It does not perform the delay itself.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters wavelength - wavelength to go to (nm)
delay - recommended delay (ms) before taking readings

C
```
int far pascal BI_select_wavelength( double     wavelength,
                                     long far   *delay );
```

Pascal
```
function BI_select_wavelength(       wavelength : double;
                               var   delay      : longint ) : integer;
```

Visual Basic
```
Function BI_select_wavelength( ByVal wavelength As Double,
                               ↳delay As Long ) As Integer
```

BI_send

Syntax BI_send(message, id)

Description This function sends a character string to the device at the specified IEEE address. It is intended to be used for communicating with devices that are not controlled by any other DLL functions (e.g. non-standard or third party hardware). It should **not** be used to communicate with devices that the DLL controls.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters message - a pointer to the string to be sent to the device
id - component identifier as set in the system configuration file

C
```
int far pascal BI_send(   char far *message, char far* id );
```

Pascal
```
function BI_send(message: pchar; id: pchar): integer;
```

Visual Basic
```
Function BI_send( ByVal message As String, ByVal id As String ) As Integer
```

BI_set

Syntax BI_set(id, token, index, value)

Description This function sets the value of the specified component attribute.

Return value The return value indicates the result of the call:
BI_OK - success
BI_invalid_token - invalid token
BI_invalid_component - the component does not exist in this system
BI_invalid_attribute - the attribute is not accessible in this system

Parameters
id - component identifier as set in the system configuration file
token - the token for attribute to be set
index - for accessing indexed attributes and setups
value - the new attribute value

C int far pascal BI_set(char far*id, int token, int index, double value);

Pascal function BI_set(id: Pchar;token: integer; index: integer;
value: double): integer;

Visual Basic Function BI_set(ByVal id As String, ByVal token As Integer,
↳ByVal index As Integer, ByVal value As Double) As Integer

Notes Using *BI_set* never triggers a hardware settle delay. If an attribute is being changed that causes some hardware operation to occur is the responsibility of the client to ensure that any required settle delay is used.

BI_set_str

Syntax BI_set_str(id, token, index, s)

Description This function sets the value of the specified component attribute, where value is a string

Return value The return value indicates the result of the call:
BI_OK - success
BI_invalid_token - invalid token
BI_invalid_component - the component does not exist in this system
BI_invalid_attribute - the attribute is not accessible in this system

Parameters
id - component identifier as set in the system configuration file
token - the token for attribute to be set
index - for accessing indexed attributes and setups
s - the new attribute string

C int far pascal BI_set(char far*id, int token, int index, char far* s);

Pascal function BI_set(id: Pchar;token: integer; index: integer;
s: Pchar): integer;

Visual Basic Function BI_set(ByVal id As String, ByVal token As Integer,
↳ByVal index As Integer, ByVal s As String) As Integer

BI_trace

Syntax BI_trace (trace)

Description This function toggles the dll tracer. When trace equals 1 the tracer will log dll function calls in C:\trace.txt

Return value None

Parameters trace : 0 – tracer off
1 – tracer on

C void BI_trace (int trace);

Pascal procedure BI_trace(trace: integer);

Visual Basic Sub BI_use_group(ByVal trace As Integer)

BI_use_group

Syntax BI_use_group(group)

Description This function sets the active group.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters group - the number of the group to make active

C int far pascal BI_use_group(int group);

Pascal function BI_use_group(group : integer) : integer;

Visual Basic Function BI_use_group(ByVal group As Integer) As Integer

BI_version

Syntax BI_version(s)

Description This function returns version information for the DLL.

Return value none

Parameters s is a pointer to a buffer that the version information is to be copied to. The version string will be no longer than 80 characters.

C void BI_version(char far *s);

Pascal procedure BI_version(s : Pchar);

Visual Basic Sub BI_version(ByVal s As String)

BI_zero_calibration

Syntax BI_zero_calibration(start_wavelength, stop_wavelength)

Description This function performs ADC offset and system dark current measurements for the active group. The DLL records the values and uses them when calculating readings in BI_measurement. To ensure accurate readings BI_zero_calibration should be called before BI_measurement is used; once before a wavelength range scan is normally sufficient.

The information is stored with the active group so it is possible to zero-calibrate each group and then swap between them at will; the DLL will always use the correct zero-calibration data.

Return value The return value indicates the result of the call:
BI_OK - success
BI_error - failure

Parameters start_wavelength – wavelength to zero calibrate from.
stop_wavelength – wavelength to zero calibrate to.

C
```
int far pascal BI_zero_calibration( double  start_wavelength,
                                    double  stop_wavelength );
```

Pascal
```
function BI_zero_calibration(  start_wavelength  : double;
                               stop_wavelength  : double ) : integer;
```

Visual Basic
```
Function BI_zero_calibration( ByVal start_wavelength As Double,
                              ↳ByVal stop_wavelength As Double ) As Integer
```

2.5. HARDWARE ATTRIBUTE TOKENS

Hardware attributes can be classified as one of two types:
i) **Structural attributes -** these describe what components the system is built from and how they interact. These attributes can be considered as hardwired.
ii) **Component attributes -** these describe how the components are set-up and used, which may differ from run to run.

The system configuration file passed to BI_build_system_model contains structural attributes. It describes the hardware system in terms of its components and how they relate to each other. Examples of this type of information are:

- there is a 228A ADC at address 29 on the IEEE bus,
- there is a TM300 monochromator that contains a filter wheel controlled by card 2 in the MSD.

The file created by BI_save_setup consists of component attributes. It describes in detail exactly how each component is configured. Examples of this type of information are:

- the ADC is sampled 10 times per measurement,
- the recommended delay time after changing filter wheel position is 2 seconds.

Once *BI_build_system_model* has been called the structural attributes cannot be changed. However, users may wish to alter component attributes such as ADC samples per reading or delay times. Component attributes are retrieved and set using the *BI_get* and *BI_set* functions.

To access an attribute with *BI_get* or *BI_set* the component id, attribute, and in some cases attribute index, must be specified. The component id is the same as that set in the system configuration file. The attribute token is a constant that the DLL recognises as referring to a particular attribute. The SDK includes a set of attribute token definition files for various languages. These give identifiers to the attribute tokens which should be used instead of the actual token values. This makes calls to *BI_get* and *BI_set* more readable and will prevent problems with future versions of the DLL where the token values may change. Table 2 gives a list of which languages are currently supported and how the attribute token definition files should be used.

The attribute index is used where an attribute token may refer to one of several values. For example, consider a 6 position filter wheel. The following function call will retrieve the filter value at position 3:

```
BI_get(     "filter_wheel",     the identifier for the filter wheel
      FWheelFilter,             attribute token for a filter value
      3,                        attribute index for position 3
      filter_value )            attribute value returned in this variable
```

Language	Attribute Token Def File	Use
C++	dlltoken.h	#include "dlltoken.h" in any source file that uses BI_get or BI_set
Pascal	dlltoken.pas	Include dlltoken.pas in any source file that uses BI_get or BI_set
Visual Basic	dlltoken.bas	Add dlltoken.bas in any project that uses BI_get or BI_set

Table 2: Attribute Token Definition Files

If there is not a token definition file for a particular programming language the file dlltoken.txt contains a no-frills list of identifiers and values that can be converted to any language. This file may be copied as required.

The following is a list of all of the attribute tokens in the DLL. There is a brief description of each attribute including how it is indexed (if it is a multi-valued attribute), what values it can take and what it defaults to. The attributes are grouped by component.

Monochromator Attributes

MonochromatorCurrentDialReading	current dial reading (M300/DM150)	0
MonochromatorCurrentWL	current wavelength (nm)	undefined
MonochromatorParkDialReading	park position (M300/DM150)	0
MonochromatorCurrentGrating	current grating (1-3)	undefined
MonochromatorScanDirection	scan direction	1
MonochromatorPark	repark monochromator	Undefined
MonochromatorSelfPark	legacy attribute (DM150)	1
MonochromatorModeSwitchNum	double – single sam index	-1
MonochromatorModeSwitchState	sam state	0
MonochromatorCanModeSwitch	boolean	1
GratingA	alpha (TM/DTM30)	1
Gratingd	ruling density d (TM/DTM300)	0
GratingWLMax	grating maximum λ (TM/DTM300)	from d
GratingWLMin	grating minimum λ (TM/DTM300)	from d
GratingX	grating constant 1 (HR600)	undefined
GratingX1	grating constant 2 (HR600)	undefined
GratingX2	grating constant 3 (HR600)	undefined
GratingZ	zord (TM/DTM300)	0
ChangerZ	changer zord	

Filter Wheel Attributes

FWheelCurrentPosition	current filter position	undefined
FWheelFilter	filter value	0
FWheelPositions	number of filter positions	undefined

SOB Attributes

SOBInitialState	state after BI_initialise	0
SOBState	current state	undefined

TLS Attributes

TLSCurrentPosition	current light source	1
TLSPOS	light source to switch to	0
TLSPositionsCommand	switch light source	undefined
TLSSelectWavelength	select wavelength for the TLS	undefined
TLSWL	wavelength to switch light source at	0

262 Attributes

biRelay	relay status	0
biCurrentRelay	current relay status	undefined

SAM Attributes

SAMInitialState	state after BI_initialise	0
SAMCurrentState	SAMs current state	undefined
SAMState	state at SAMSwitchWl(n)	0
SAMSwitchWL	state change wavelength (nm)	undefined
SAMDeflectName	name for deflected SAM state	Deflect
SAMNoDeflectName	Name for undeflected SAM state	No Deflect

Stepper SAM Attributes

SSEnergisedSteps	steps to energised position	undefined
SSRelaxedSteps	steps to relaxed position	undefined
SSMaxSteps	maximum steps	undefined
SSSpeed	motor speed	undefined
SSMoveCurrent	motor current on move	undefined
SSIdleCurrent	motor current when idle	undefined

MVSS Attributes

MVSSConstantBandwidth	current constant Bandwidth (nm)	undefined
MVSSConstantwidth	current constant width (mm)	0..10
MVSSCurrentWidth	current width of the mvss (mm)	undefined
MVSSPosition	set slit position in monochromator (‘entrance’, ‘exit’ or ‘middle’)	‘entrance’
MVSSSetWidth	move the slit to the specified width (mm)	0..10
MVSSSlitMode	the current slit drive mode	0 / 1
MVSSSwitchWL	state change wavelength (nm)	undefined
MVSSWidth	width at specified state	0..10

ADC Attributes

ADCAdaptiveIntegration	the adaptive integration mode	0/1
ADCSamplesPerReading	number of samples integrated for one reading	10
ADCSamplePeriod	sample period in milliseconds	100
ADCVolts	take a reading	undefined
ADCAuxVolts	read from a 487 Aux Input	undefined
ADCAuxOffset	get/set the Aux Offset	0
ADCAuxInput	select the 487 Aux Input	undefined
ADCTimeConstant	for use with Stanford lock-in	undefined
ADCXYThetaReading	for use with Stanford lock-in	undefined

General Amplifier Attributes

AmpChannel	input channel (225, 267, 277 only)	1
AmpCurrentChannel	current channel (225, 267, 277 only)	undefined
AmpCurrentRange	current range	undefined
AmpGain	gain value	
AmpMinRange	minimum range	1
AmpMaxRange	maximum range	7 (225) 6 (others)
AmpOverload	overload flag	0
AmpOverrideWl	set group readings to 1 from this wavelength (nm)	
AmpStartRange	starting range (265, 267, 277 only)	1
AmpUseSetup	λ (nm) to switch to set-up (225, 267, 277 only)	0
AmpCurrentSetup	get/set current setup being used	0

225 Attributes

A225fMode	frequency mode	1
A225PhaseQuadrant	phase quadrant	1
A225PhaseVariable	phase variable	0
A225TargetRange	target range	1
A225TimeConstant	time constant	1

Motorised Stage Attributes

MotorPosition	get/set motor position	0
MotorStop	tell motor to stop moving	undefined

EBox Monitor Attributes

EBoxReadHV	Read High Voltage	Undefined
EBoxReadTemp	Read Temperature	Undefined
EBoxReadHVRaw	Read High Voltage without converting	Undefined
EBoxReadTempRaw	Read Temperature without converting	Undefined
EBoxWait	Delay between HV or Temp readings	5000
EBoxRepeats	Repeats of HV and Temp	1
EBoxCountsAtTargetTemp	ADC reading at target temp	11990
EBoxGradientTemp	Gradient of ADC counts vs Temp	1288.19
EBoxTargetTemp	Target Temperature	23
EBoxCountsAtTargetHV	ADC reading at target HV	10178
EBoxGradientHV	Gradient of ADC counts vs HV	-216.76
EBoxTargetHV	Target High Voltage	750

Camera Attributes

CameraIntegrationTime	Get/Set Integration Time	Undefined
CameraBeta	Get/Set Camera Beta	0
CameraPhi	Get/Set Camera Phi	0

Miscellaneous Attributes

biSettleDelay	recommended settle delay (ms) after operation	
biMin	minimum allowable position (Motor)	-1
biMax	maximum allowable position (Motor)	-1
biParkPos	position after parking (M300, DM150, Motor only)	-1
biInput	input channel indexed by setup (262, 265, 267, 277)	
biCurrentInput	current input channel (262, 265, 267, 277)	
biMoveWithWavelength	change component state with wavelength	
biHasSetupWindow	ask whether setup window exists for component	
biHasAdvancedWindow		
biDescriptor	returns product description string (eg DTM300)	
biProductName	returns product name (eg DTM300 Monochromator)	
biParkOffset	park offset	

System attributes

Sys225_277Input	which 225 input the 277 output goes to	1
SysStopCount	stop-count value for AC zero-calibration	1.0
SysDarkIIntegrationTime	Integration time to be used when calculating dark levels	

Bentham Hardware Types

BenInterface	hardware_type token for interfaces
BenSAM	hardware_type token for SAMs

BenSlit	hardware_type token for MVSSs
BenFilterWheel	hardware_type token for filter wheels
BenADC	hardware_type token for ADCs
BenPREAMP	hardware_type token for preamplifiers
BenACAMP	hardware_type token for ac amplifiers
BenDCAMP	hardware_type token for dc amplifiers
BenPostAMP	hardware_type token for post amplifiers
BenRelayUnit	hardware_type token for relay units
BenMono	hardware_type token for monochromators
BenAnonDevice	hardware_type token for anonymous devices
BenCamera	hardware_type token for camera devices
BenDiodeArray	hardware_type token for diode array devices
BenORM	hardware_type token for ORM400 devices
BenEBoxMonitor	hardware_type token for EBoxMonitor device
BenUnknown	hardware_type token for unknown devices

The use of attribute tokens in controlling components is explained in detail further on.

2.6. DLL ERROR CODES

Functions in the DLL return error codes to indicate their success or failure and the cause of any failure. These are defined as follows:

BI_OK - function call succeeded
BI_error - function call failed
BI_invalid_token - the function was passed an invalid attribute token
BI_invalid_component the function was passed a component identifier that does not exist
BI_invalid_attribute - the function was passed an attribute token referring to an attribute that does not exist or is inaccessible

Hardware operation functions return either BI_OK or BI_error. When a hardware operation function returns BI_error, BI_report_error can be used to return a hardware error code that describes exactly what caused the error. The hardware error codes are as follows:

BI_no_error - no error to report
BI_PMC_timeout - PMC not responding
BI_MSD_timeout - MSD not responding
BI_MSC_timeout - MSC1 not responding
BI_MAC_timeout - MAC not responding
BI_MAC_invalid_cmd - error in communication with MAC
BI_225_dead 225 not responding
BI_265_dead - 265 not responding
BI_267_dead - 267 not responding
BI_277_dead - 277 not responding
BI_262_dead - 262 not responding
BI_ADC_read_error - ADC not responding
BI_ADC_invalid_reading could not obtain valid ADC reading
BI_AMP_invalid_channel invalid amplifier channel
BI_AMP_invalid_wavelength invalid amplifier wavelength
BI_SAM_invalid_wavelength invalid SAM wavelength
BI_MVSS_invalid_width invalid MVSS wavelength
BI_turret_invalid_wavelength -attempt to send monochromator beyond wavelength range
BI_turret_incorrect_pos -error in communication with MAC

Calling *BI_report_error* clears the error code, i.e. subsequent calls will return *BI_no_error* until another hardware error occurs.

The SDK includes error code definition files. These should be used in applications that need to know what the return values of DLL functions signify. Languages currently supported are:

Language	Error Code Definition File	Use
C	dllerror.h	Use #include "dllerror.h" in any source file that uses error codes
Pascal	dllerror.pas	Include dllerror.pas in any source file that needs to know the error codes
Visual Basic	dllerror.bas	Add in any project that uses error codes

Table 3: Error Code Definition Files

If there is not an error code definition file for a particular programming language the file dllerror.txt contains a no-frills list of identifiers and values that can be converted to any language. This file may be copied as required

2.7. CONTROLLING HARDWARE VIA THE DLL

This section describes how attribute tokens are used to control the behaviour of the components in a system.

2.7.1. M300/Mc300 and DM150/DMc150 Monochromators

The DLL is designed to allow monochromator control at the highest level; the function *BI_select_wavelength* tells the DLL which wavelength the monochromator should be at and the DLL does the rest, co-ordinating the operation of the gratings and any filter wheels or SAMs. Before this can happen the DLL needs to know exactly how the monochromator is set up.

MonochromatorCurrentDialReading: the current dial reading. For non-parking M300 and DM150 monochromators this **must** be set before any calls to *BI_select_wavelength*. Valid values are 0..999.99.

MonochromatorCurrentWavelength: the wavelength (nm) that the monochromator is at; read-only.

MonochromatorScanDirection: controls when anti-backlash precautions are taken. A value of 1 indicates that all wavelengths should be approached from a shorter wavelength, while 0 indicates that all wavelengths should be approached from a longer wavelength. If the DLL ever needs to approach a wavelength in the 'wrong' direction it will first overshoot the required position and then go to it from the other side, preventing any backlash error. Therefore if an application intends to scan from short to long wavelengths (e.g. UV to IR), MonochromatorScanDirection should be set to 1. If an application intends to scan from long to short wavelengths (e.g. IR to UV), MonochromatorScanDirection should be set to 0. The default value is 1 (increasing wavelength).

Gratingd: the line density *d* (lines mm^{-1}) of the grating.

GratingWLMax: the maximum wavelength (nm) that the grating should be used for. Setting this to 0 causes the DLL to use a default value chosen by the ruling density.

GratingWLMin: the minimum wavelength (nm) that the grating should be used for. Setting this to 0 causes the DLL to use a default value chosen by the ruling density.

biSettleDelay: the recommended settle delay after selecting a new wavelength. *BI_select_wavelength* will return this value if the monochromator has changed position and it is the longest delay time triggered.

MonochromatorPark: called with any value, the monochromator will be reparked.

MonochromatorSelfPark: legacy token for telling the benhw whether a Dial Reading is entered by the user when the monochromator park is called, or whether the monochromator parks itself.

2.7.2. TM300/TMc300, DTM300/DTMc300 and TTM300 Monochromators

As with the M300 and DM150 monochromators, the DLL needs to know the set-up for TM300 and DTM300 monochromators. For a TM300 gratings are indexed as 1, 2 or 3. For a DTM300 gratings on turret 1 are indexed as 11, 12 or 13, and on turret 2 as 21, 22 or 23.

MonochromatorCurrentWavelength: the wavelength (nm) that the monochromator is at. This attribute is read-only. Attempting to set it *will not* operate the monochromator; use the DLL function *BI_select_wavelength*.

MonochromatorScanDirection: controls when anti-backlash precautions are taken. A value of 1 indicates that all wavelengths should be approached from a shorter wavelength, while

0 indicates that all wavelengths should be approached from a longer wavelength. If the DLL ever needs to approach a wavelength in the 'wrong' direction it will first overshoot the required position and then go to it from the other side, preventing any backlash error. Therefore if an application intends to scan from short to long wavelengths (e.g. UV to IR), MonochromatorScanDirection should be set to 1. If an application intends to scan from long to short wavelengths (e.g. IR to UV), MonochromatorScanDirection should be set to 0. The default value is 1 (increasing wavelength).

GratingA: the alpha value for a grating.

Gratingd: the line density *d* (lines mm^{-1}) for a grating.

GratingWLMax: the maximum wavelength (nm) that the grating should be used for.

GratingWLMin: the minimum wavelength (nm) that the grating should be used for.

GratingZ: the zero order (zord) value for a grating.

biSettleDelay: the recommended settle delay after selecting a new wavelength. *BI_select_wavelength* will return this value if the monochromator has changed position and it is the longest delay time triggered.

2.7.3. Filter Wheel

Filter wheels may have 6, 8, 10 or 12 positions; 6 is standard. The number of positions is fixed in the system configuration file. Each position can be assigned a filter value. If the filter wheel is attached to a monochromator then a call to *BI_select_wavelength* will send the filter wheel to the position with the highest filter value that is less than or equal to the specified wavelength.

The last position is always used as the shutter position, but this may also be assigned a filter value. *BI_close_shutter* will send the filter wheel to this position. Positions with no filters should have a filter value of 0 (default). The filter values do not have to be assigned in any particular order, but normal practice is to start with the lowest filter and work up.

FWheelCurrentPosition: the current filter position. This *must* be set for non-parking M300 and DM150 systems before calling *BI_zero_calibration*, *BI_select_wavelength* or *BI_close_shutter*. Read only for MAC and MSD controlled monochromators

FWheelFilter: filter value, indexed by position (1..number of positions). This value is used to select a filter when *BI_select_wavelength* is called. When there is no filter at position *n* FWheelFilter(*n*) should be 0 (default).

FWheelPositions: specifies the number of filter positions. The last (highest numbered) position is always used as the shutter. Read only.

biSettleDelay: the recommended settle delay after a change of filter position. *BI_select_wavelength* will return this value if the filter wheel has been moved and it is the longest delay time triggered.

2.7.4. SAM

SAMs can be in one of two defined states, energised or relaxed. SAMs have an initial state that they go to when *BI_initialise* is called and also maintain a list of up to 10 records that specify what state the SAM should be in at a given wavelength. If a SAM is attached to a monochromator then a call to *BI_select_wavelength* will cause the SAM to go to the state corresponding to the record with the highest wavelength that is less than or equal to the wavelength specified.

SAMInitialState: the state of the SAM after calling *BI_initialise*. 0 indicates relaxed and 1 is energised. The default is 0 (relaxed).

SAMSwitchWl, SAMState: used to access the SAM's wavelength-state records. Both attributes are indexed by the setup number (1..10). *SAMSwitchWl* refers to the wavelength and *SAMState* to the corresponding state.

SAMDeflectName, SAMNoDeflectName: used to read/write the SAM state names. For example the state may be given the name of the detector the light will fall upon.

biSettleDelay: the recommended settle delay after the SAM has changed state. *BI_select_wavelength* will return this value if the SAM has changed state and it is the longest delay time triggered.

2.7.5. TLS

TLSs can be in one of three positions. TLSs have an initial position that they go to when *BI_initialise* is called and also maintain a list of up to 10 records that specify what position the TLS should be in at a given wavelength. If a TLS is attached to a monochromator then a call to *BI_select_wavelength* will cause the TLS to go to the position corresponding to the record with the highest wavelength that is less than or equal to the wavelength specified.

TLSCurrentPositon: the position that the TLS is currently in (1..3).

TLSWl, TLSPOS: used to access the TLS's wavelength-position records. Both attributes are indexed by the setup number (1..10). *TLSWl* refers to the wavelength and *TLSPOS* to the corresponding position.

biSettleDelay: the recommended settle delay after the TLS has changed state. *BI_select_wavelength* will return this value if the TLS has changed state and it is the longest delay time triggered.

2.7.6. MVSS

The width of an MVSS can be controlled be controlled in 2 ways. The first is in a constant Bandwidth more and the second in constant width mode. In constant Bandwidth mode the desired bandwidth is set and the dll controls the slit to give the bandwidth required for the current grating. Calls to BI_Select_Wavelength that result in a grating change will automatically change the slits. In Constant Width mode the Slits are set to a required width in mm. The commands are:

MVSSConstantBandwidth: this allows for the bandwidth to be set in nm.

MVSSConstantwidth : this allows for the width to be set in mm

MVSSSlitMode : this set which mode the slits should operate in (mm = 0) (nm = 1)

2.7.7. SOB

SOBs can be in one of two defined states, energised or relaxed. SOBs have an initial state that they go to when *BI_initialise* is called.

SOBInitialState: the state of the SOB after calling *BI_initialise*. 0 is relaxed and 1 is energised. The default is 0.

SOBState: the current state of the SOB. 0 is relaxed and 1 is energised.

biSettleDelay: the recommended settle delay after the SOB has changed state.

2.7.8. 228, 228A, 485 and 487 ADCs

The number of samples averaged to obtain a reading is sets the integration time. For a 228A that produces a new sample every 100ms setting 10 samples per reading means that each reading takes 1s. Increasing the integration time increases the signal-to-noise ratio. Adaptive integration makes use of this by increasing the integration time as the signal weakens.

ADCSamplesPerReading: how many ADC samples should be averaged per measurement. Indexed 0-6, where 0 is used for non-adaptive integration and 1-6 corresponds to current amplifier range (minimum-maximum gain, i.e. strong-weak signal).

ADCAdaptiveIntegration: 0 and 1 turn adaptive integration off and on respectively.

ADCSamplePeriod: the number of milliseconds between samples. Read only.

ADCVolts: this will return a reading from the ADC in volts. One of few tokens which talk directly to hardware.

ADCAuxInput: Used to set a 487 to the Auxiliary Input.

ADCAuxOffset: by default, no offset is set for the Aux Input so this must be manually taken and set.

ADCAuxVolts: this will take a reading of the 487 Auxiliary Input and return a value in volts.

2.7.9. 225, 265, 267, 277, 485,487 and 477 Amplifiers

The 225, 267, 277, 485,487 and 477 amplifiers can all be pre-programmed with multiple set-ups. The 225 and 485 have four set-ups, corresponding to four different detectors, and the 267, 277, 487 and 477 have two set-ups.

The 267/277/487/477 set-ups consist of minimum, maximum and start ranges, input channel and the wavelength from which the set-up should be used. With the 267 and 277 the set-ups are mainly used to change channel at a given wavelength.

The 225 and 485 set-ups consists of minimum, maximum and target ranges, time constant, phase quadrant and variable, frequency mode, input channel and the wavelength from which the set-up should be used. With the 225 and 485 each set-up corresponds to a different detector, and switching between set-ups provides a quick and easy way of configuring the 225 and 485 for all of the different detectors that may be encountered in a system.
When an amplifier is in the active group a call to *BI_initialise* will cause the amplifier to configure itself using the current setup. In addition *BI_select_wavelength* will cause the amplifier to use the set-up with the highest wavelength that is less than or equal to the wavelength specified.. Amplifier settings may also be 'manually' changed via *BI_set*. This clears AmpCurrentSetup.

AmpGain: the gain for a specific range. Indexed by the range number (1-6 for 265, 267 and 277, 1-7 for 225), where 1 is the least sensitive range. The default values for ranges from 1 up are:

265: 10^4, 10^3, 10^2, 10, 1, 10^{-1}
267: 10^4, 10^3, 10^2, 10, 1, 10^{-1}
277: 10^7, 10^6, 10^5, 10^4, 10^3, 10^2
225: 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}, 10^{-8}

These values are used by the DLL when calculating results for *BI_measurement* and should not normally have to be altered.

AmpChannel: input channel for a specified set-up, indexed by the set-up number. Legal values are 1 and 2; the default is 1.

AmpMinimumRange: minimum range to be used for a specified set-up, indexed by the set-up number. Legal values are 1..number of ranges, where 1 is the least sensitive range. The default is 1. Setting this to the same value as AmpMaximumRange prevents auto-ranging.

AmpMaximumRange: maximum range to be used for a specified set-up, indexed by the set-up number. Legal values are 1..number of ranges, where 1 is the least sensitive range. The default is 1. Setting this to the same value as AmpMinimumRange prevents auto-ranging.

AmpStartRange: (265, 267 and 277 only) the start range to be used for a specified set-up, indexed by the set-up number. *BI_initialise* sets the amplifier to the start range for the current set-up. Legal values are 1..number of ranges, where 1 is the least sensitive range. The default is 1.

AmpCurrentSetup: (267, 277 and 225 only) the current set-up. Legal values are 1 and 2 for 267 and 277, 1..4 for 225. This attribute is set to 0 if *AmpCurrentRange* or *AmpCurrentChannel* are accessed using *BI_set*.

AmpCurrentRange: the current range. Legal values are 1..6 for the 265/267/277, 1..7 for the 225.

AmpCurrentChannel specifies the current input channel. Legal values are 1 and 2.

AmpUseSetup: (267, 277, 225 only) the wavelength from which the set-up for detector *n* should be used, indexed by set-up number. If more than one set-up is specified for a particular wavelength then the one with the lowest value of *n* is used. The default is 0 (0nm).

biSettleDelay: the recommended settle delay after the amplifier has changed one of its settings. *BI_select_wavelength* will return this value if the amplifier has changed range or set-up and it is the longest delay time triggered.

The following apply only to the 225:

A225TargetRange: the target range to be used for a specified set-up, indexed by the set-up number. Legal values are 1..7, where 1 is the least sensitive range. The default is 1.

A225PhaseQuadrant: the phase quadrant to be used for a specified set-up, indexed by the set-up number. Legal values are 1..4, corresponding to phase quadrants of 0°, 90°, 180° and 270°. The default is 1 (0°).

A225PhaseVariable: the phase variable to be used for a specified set-up, indexed by the set-up number. The phase variable is a real number in the range 0..102.4. The default is 0.

A225TimeConstant: the time constant to be used for a specified set-up, indexed by the set-up number. Legal values are 1..7, corresponding to time constants of 10ms, 30ms, 0.1s, 0.3s, 1s, 3s and 10s. The default is 1 (10ms).

A225fMode: the frequency mode to be used for a specified set-up, indexed by the set-up number. Legal values are 1 and 2, corresponding to frequency modes f and 2f. The default is 1 (f).

2.7.10. Motorised Stages

A Motorised Stage is simply a single motor, often used for positioning in goniometer systems.

MotorPosition: used to either get the current position, or tell the motor to move to a new position. Value given is in steps.

MotorStop: when MotorPosition attribute is set and passed an index of 1, the motor is not polled to work out when the motor has finished moving, giving the user an opportunity to call MotorStop if the motor needs to stop prematurely.

2.7.11. EBox Monitor

EboxReadHv: The High Voltage value, converted to volts using the gradient HV, target HV and counts at Target HV attributes.

EboxReadTemp: The Temperature value, converted to degrees celsius using the gradient temp, target temp and counts at target temp attributes.

EboxReadHvRaw: A single unconverted HV reading in ADC counts.

EboxReadTempRaw: A single unconverted temperature reading in ADC counts.

EboxWait: The waiting period between repeated readings of the HV or temperature values.

EboxRepeats: The number of repeats to perform for HV and temperature readings.

EboxCountsAtTargetTemp: The ADC counts at the given target temperature.

EboxGradientTemp: The gradient of the ADC counts vs temperature line.

EboxTargetTemp: The target temperature value in degrees celsius.

EboxCountsAtTargetHv: The ADC counts at the given target high voltage.

EboxGradientHv: The gradient of the ADC counts vs high voltage line.

EboxTargetHv: The target high voltage value in volts.

2.7.12. Camera Attributes

CameraAutoRange: Whether the Camera automatically changes integration time based on the maximum value across the array during a measurement.

CameraIntegrationTime: The Camera integration time in milliseconds. Can be retrieved or set as appropriate.

CameraBeta: The Camera Beta. This value is used along with Phi to calculate the wavelength scale across the camera array.

CameraPhi: The Camera Phi. This value is used along with Beta to calculate the wavelength scale across the camera array.

2.7.13. System Attributes

These attributes do not belong to any particular components, but control how the system model takes measurements. In particular these attributes can be used to set-up adaptive integration in DC systems. This can speed up measurements by allowing a short integration time to be used for high signals, where the signal-to-noise ratio is already very large, and longer integration times for low signals, where the signal-to-noise ratio needs to be improved.

SysStopCount: the autozero stop-count value for AC systems. When the zero offset is being calculated for an AC system (*BI_zero_calibration*) the DLL samples the ADC until the difference between two consecutive readings is equal to or less than this value. This ensures that the system has settled. The default is 1.

SysDarkIntegrationTime: the integration time (s) for the dark current reading in DC systems (AC systems have no dark current). The default is 5 (5s); this is equivalent to 50 samples with a 228A. Calculating the dark current is part of the zero calibration routine (*BI_zero_calibration*).

Sys225_277Input: the input on the 225 that the output from the 277 is connected to. The DLL uses this information to work out which combination of amplifiers is being used when it calculates the result for *BI_measurement*. The default is 1 (i.e. 277 output to 225 input 1).

2.8. USING THE DLL

This section gives hints and tips on using the DLL with different programming languages and packages. As a rough guide, an application would normally call the DLL functions in the following order:

DLL start-up and hardware initialisation

i) BI_build_system_model
ii) BI_load_setup
iii) BI_initialise
iv) BI_park
v) BI_get / BI_set (if required)

Measurement cycle

vi) BI_zero_calibration
vii) BI_select_wavelength (pause using returned settle delay)
viii) BI_automeasure
ix) return to step vii)

When a DLL is loaded into memory it does not have its own stack for function calls and local variables but uses the client application's stack instead. Although the spectroradiometer control DLL is not particularly stack-hungry this should be noted in case any stack errors do occur. In normal use a client stack size of 8Kb has been found sufficient (depending on how much the client uses, of course).
The serial port used by the DLL to communicate with the hardware is set in the system configuration file (\hardware\system.cfg on the SDK disk). If you wish to use a different serial port then you will need to edit the line

```
TAS016 comms port Comn
```

so that COMn refers to the port that you wish to use.

The file bendll.h contains declarations for the functions in the DLL. Any source files that use functions from the DLL should include the following line:

```
#include "bendll.h"
```

The file dlltoken.h is the C/C++ token definition file. Any source files that use BI_get or BI_set need to know about attribute tokens. This is done by making sure that they contain the line:

```
#include "dlltoken.h"
```

The file dllerror.h is the C/C++ error code definition file. Any source files that need to know about error codes should contain the line:

```
#include "dllerror.h"
```

When creating a module definition file for your application a stack size of no less than 8Kb should be specified otherwise you may encounter stack errors.
The same as for C (see previous section) but the macro `__cplusplus` must be defined to ensure that the DLL functions are declared correctly (some compilers do this automatically).
The file dlltoken.pas is the Pascal attribute token definition file. This should be included in any source files that use BI_set or BI_get (please refer to your compiler manual for instructions on how to do this). The file dllerror.pas is the Pascal error code definition file. This should be included in any files that need to know about error codes. Your compiler should also be told to allocate at least 8Kb to the stack or you may encounter stack errors.
The Visual Basic attribute token definition file and error code definition file are dlltoken.bas and dllerror.bas respectively; these need to be included in any project using the DLL. In addition the file bendll.bas contains declarations of all of the DLL functions. Including this as a

code module in a project ensures that the DLL functions are available to all other code and form modules in the project.

3. TROUBLESHOOTING

3.1. WINDOWS ERRORS

These are error messages that Windows may report when it encounters a problem in running an application using the spectroradiometer control DLL. They appear in dialogue boxes that lock the system until dismissed.

File Error. Unable to find <name>.DLL.
Ensure that <*name*>.dll is correctly installed.

Error. Runtime error 202 at xxxx:xxxx.
This is a stack overflow error; for details on DLLs and the stack see Using The DLL.

Error. Runtime error 006 at xxxx:xxxx.
This error can occur when an application using the DLL has crashed and is re-executed. When an application crashes Windows does not unload the DLL and it is left in memory in an undefined state. Trying to then access the DLL again will almost always fail, producing this error. If your application crashes the safest course of action is to restart Windows; unfortunately there is no other way of removing the DLL from memory.

3.2. HARDWARE CONTROL PROBLEMS

All of the DLL functions return BI_error and the hardware will not respond.
Ensure that you have called BI_build_system_model and that it has succeeded (i.e. returned BI_OK).

BI_initialise keeps failing
Ensure that you have called BI_build_system_model and that it has succeeded (i.e. returned BI_OK). For IEEE TM300/DTM300 systems ensure that all cable connections are secure and that the hardware is switched on.

The monochromator does not go to the correct wavelength.
For TM300/DTM300 systems check:
- You have called BI_initialise and it has succeeded,
- You have called BI_park and it has succeeded,
- The DLL has the correct z-ord and α values for your monochromator.

For non-parking M300/DM150 systems check:
- The DLL has the correct dial reading.

For self parking M300/DM150 systems check:
- You have called BI_park and it has succeeded.

The filter wheel does not go to the correct position.
For TM300/DTM300 systems check:
- You have called BI_initialise and it has succeeded,
- You have called BI_park and it has succeeded.

For non-parking M300/DM150 systems check:
- The DLL has the correct filter position.

For self-parking M300/DM150 systems check:
- You have called BI_park and it has succeeded,

For all systems check:

•The DLL has the correct filter value for each position (any empty positions should be set to a filter value of 0).

BI_autorange keeps failing.

BI_measurement will fail if there is a problem reading the ADC. This is often caused by a communications error or a sudden change in the input to the ADC.

•An intermittent communications error can be caused by a bad connection on the IEEE bus; check the cable connections.

•The ADC can experience a sudden change in input when the amplifier(s) is(are) ranged or the filter wheel changes position. Ensure that all delays are long enough and are actually used.

BI_zero_calibration keeps failing.

BI_zero_calibration will fail for the same reasons as BI_autorange; refer to the previous answer.

BI_measurement keeps failing.

BI_measurement will fail for the same reasons as BI_autorange; refer to the previous answer.