

B • R • U • S • A • G

Sensorik & messtechnische Entwicklungen AG

Chapfswiesenstrasse 14

CH-8712 Stäfa (Switzerland)

INTRA

Software Interface Definition Document

(SIDD)

BRUSAG Ref: INTRA/SIDD/1827-BRU

Version 1.02 of 03-Mar-2010

Change Record

Date	Version	Who	Description
08-Jul-08	1.00	RB	Draft started
02-Jul-09	1.01	RB	doc updated according to existing soft- and firmware resp.
03-Mar-10	1.02	RB	new parameter sun2rad in IROM – which now gets version 0x101

Table of Contents

1	Introduction	1
1.1	Scope of the document	1
1.2	References	1
2	Remote Procedures	1
2.1	Overview of Procedures	1
2.2	Data Structures and Protocol	2
2.3	The transport protocol EDLP	3
2.4	Detailed Description of Procedures	4
2.4.1	WhoAmI	5
2.4.2	Set ROMP	5
2.4.3	GetROMP	6
2.4.4	ROMPrw	6
2.4.5	SetDateTime	7
2.4.6	GetDateTime	7
2.4.7	SetMode	7
2.4.8	GetMode	7
2.4.9	SetPos	8
2.4.10	GetPos	8
2.4.11	GetSun	9
2.4.12	GetMem	9
2.4.13	SetMem	9
2.4.14	FindZero	10
2.4.15	ChkAxis	10
2.4.16	GetLog / Clearlog	10
2.4.17	RunMotors	11
2.4.18	GetADC	11
2.4.19	SetLogMode	11
3	Glossary	12

1 Introduction

1.1 Scope of the document

This document defines the software interface between INTRAs controller and a PC resident software. The interface is based on remote procedure calls. We will use and follow as closely as possible – the standards set for remote procedure calls in and by various RFCs.

Our first implementation of the RPC mechanism for the new controller will however not use TCP/IP as transport mechanism, but a much simpler protocol used for serial links. We already used this protocol¹ before for INTRA (firmware version 2.48) and more recently for MetNevis.[1]

1.2 References

ID	Title	Version & Date	Doc-ID
1	MetNevis – Software Interface Definition Document R. Brusa, BRUSAG CH-8712 Stäfa	18-Sep-06	SNOW/SIDD/ 427-BRU
1	AT91 ARM Thumb-based Microcontrollers AT91SAM7X512, AT91SAM7X256, AT91SAM7X128	08-Oct-07	6120G–ATARM
2	Circuit Drawing INTRA Controller Board V2	June 2008	-
3	INTRA Software Definition Document (SDD) R. Brusa, BRUSAG CH-8712 Stäfa	V 1.00 2-Jul-08	INTRA/SDD/ 1826-BRU
4	INTRA Software Interface Definition Document (SIDD) R. Brusa, BRUSAG CH-8712 Stäfa (this doc)	V 1.02 03-Mar-10	INTRA/SIDD/ 1827-BRU
5	Explanatory Supplement to the Astronomical Almanac; Seidelmann, P. Kenneth, Ed. University Science Books Mill Valley CA 94941	1992	ISBN 0-935702- 68-7
6	RPC: Remote Procedure Call Protocol Specification Version 2.; R. Srinivasan, Status: PROPOSED STANDARD	August 1995	RFC 1831
7	XDR: External Data Representation Standard; R. Srinivasan. Status: DRAFT STANDARD	August 1995	RFC 1832
8	Binding Protocols for ONC RPC Version 2; R. Srinivasan. Status: PROPOSED STANDARD	August 1995	RFC 1833
9	Authentication Mechanisms for ONC RPC; A. Chiu. Status: INFORMATIONAL	September 1999	RFC 2695

2 Remote Procedures

2.1 Overview of Procedures

Name	Description
<i>WhoAmI</i>	returns firmware-identifying string
<i>SetROMP</i>	upload rom-parameters to ram
<i>GetROMP</i>	download rom-parameters from ram
<i>ROMPrw</i>	rom parameters ram_copy r<-->w erom
<i>SetDateTime</i>	set RTC - date and time of day

¹ EDLP ESTEC Data link protocol

GetDateTime	get date and time of day from RTC
SetMode	set mode INIT, SUN, CLOCK or REMOTE
GetMode	get current mode and submode
SetPos	set new targets
GetPos	get mode and positions (PA,SA) and (az,el)
GetSun	get AD-readings of q0..q3 of sun sensor
GetMem	get <u>nb</u> bytes starting at <u>adr</u>
SetMem	write to memory 1, 2 or 4 bytes
FindZero	start search zero mark on specified axis
ChkZero	get status of zero search (not impl. - see ChkAxis)
ChkAxis	get status of both axis
GetLog	get <u>n-th</u> line of log or clear log.
RunMot	disable/enable normal operation, run motors in test-mode
GetADC	get all ADC-signals in mode raw, volt or phys
SetLogMode	set Log-mode

2.2 Data Structures and Protocol

The remote procedure calls are all implemented for the program ifw – the firmware that currently runs in INTRA's controller. Note that the controller is an arm processor and as such operated using the little endian convention. On the other hand, RPC specifications require big endian. This will force us to perform some byte inversions.

In line with RPC1831, the program definition for this rpc-application could be written as follows:

```

program ifw {
    /*
     * Latest and greatest version
     */
    version INTRA_RPC {
        void WhoAmI(string progname, int v) = 0;
        void SetRomE(EEPROM prom) = 1;
        /*
         * write prom-data into rom-proxy area in RAM
         */
        void GetRomP(EEPROM* promp) = 2;
        .....and more
    } = 1;
}

```

The arguments of the call of a procedure are packed into a structure of type `rpc_msg` as follows:

```

struct rpc_msg {
    unsigned int xid; // counter to identify call
    union switch (msg_type mtype) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};

```

The above and all following definitions in this paragraph follow RFC 1832. We briefly repeat the essentials of these definitions:

- Data are big endian – the MSB precedes the LSB – (but remember: our controller is little endian)
- All data must be assembled (and eventually padded with zeros) into multiples of 4 Bytes.
- The xid of a REPLY must match the xid of the corresponding call.

The structure of the call body is shown below:

```
struct call_body {
    unsigned int rpcvers;          /* must be equal to two (2) */
    unsigned int prog;            /* 20000000h...3fffffffh use 23456789h
                                */
    unsigned int vers;            /* of remote prog */
    unsigned int proc;           /* identifies procedure */
    opaque_auth cred;            /* credentials : use AUTH_NONE = 0 */
    opaque_auth verf;           /* verification: use AUTH_NONE */
    /* procedure specific parameters start here */
};
```

and the reply-body is shown here:

```
union reply_body switch (reply_stat stat) {
    case MSG_ACCEPTED:
        accepted_reply areply;
    case MSG_DENIED:
        rejected_reply rreply;
} reply;
```

and further expanded as:

```
struct accepted_reply {
    opaque_auth verf;
    union switch (accept_stat stat) {
        case SUCCESS:
            opaque results[0];
            /*
             * procedure-specific results start here
             * use struct rProcName {..}
             */
        case PROG_MISMATCH: // between program of caller and target
            struct {
                unsigned int low;
                unsigned int high;
            } mismatch_info;
        default:
            /*
             * Void. Cases include PROG_UNAVAIL, PROC_UNAVAIL,
             * GARBAGE_ARGS, and SYSTEM_ERR.
             */
            void;
    } reply_data;
};
```

```
union rejected_reply switch (reject_stat stat) {
    case RPC_MISMATCH:
        struct {
            unsigned int low;          /* lowest protocol version = 2 */
            unsigned int high;        /* highest is 2 as well */
        } mismatch_info;
    case AUTH_ERROR:
        auth_stat stat;
};
```

2.3 The transport protocol EDLP

For transmission, a RPC data structure is framed between stx and etx. A byte stuffing mechanism ensures that these two characters do never occur in the transmitted characters. Byte-stuffing for transmission is as follows:

Data		Substitution	
dle	10h	dle 'D'	10h, 44h
stx	02h	dle 'S'	10h, 53h
etx	03h	dle 'E'	10h, 45h

During transmission, a checksum – the sum of all characters of the original packet is computed. The two's complement of this value – possibly with byte stuffing – is then inserted in the data stream and the packet is closed by sending the terminating etx.

Upon reception, the leading stx-character is dropped and the data stream is destuffed according to the above table. A packet is considered complete when the terminating etx-character is received. The checksum that was previously computed from the destuffed data must now be zero, otherwise, it is considered a checksum error.

A packet with a checksum error is silently trashed. Furthermore – an unexpected stx received during reception of a packet causes the current packet to be silently trashed and a new packet is started.

Transmission is initiated by the host only. If the server does not answer within a specified interval of time, the transmission is repeated 3 times and then a communication timeout error is reported (global flag).

Note that this protocol allows for normal terminal I/O on the same lines as the remote procedure calls. Everything that is not framed between stx/etx is considered normal terminal I/O. The current INTRA firmware uses this to output diagnostics messages.

2.4 Detailed Description of Procedures

Prior to go into full depth, a remark is in order: IntraCfg.exe is a utility that is available from our web site. It allows a user to communicate with and configure an INTRA for use without having to deal with all these RPC-issues. However, if you want to integrate tracker-handling routines into your own code, have a closer look at the source code of IntraCfg. It implements all these routines on a PC and its source can also be found on our web site. IntraCfg was written with Delphi 7.2.

Ok, back to the procedures. All procedure return an integer value of 0 if no errors were encountered. The interpretation of a non-zero result is according to the following enumeration:

```
EStatus = (ok, argerr, chkserr, cmderror, verserror, iderror,
           bccerror, rtimeout, rpending);
```

where

ok: no error
 argerr there was a problem with one or more arguments
 chserr irrevocable errors during transmission (either way)
 cmderr version error (not used)
 iderror id-error (not used)
 bccerror mismatch of byte-count occurred
 rtimeout no answer within specified interval of time
 rpending waiting for answer

In the following description we use a notation that deviates somewhat from standard C in that each argument gets an attribute: in, out or var. Clearly, in-arguments are part of the call only, out-arguments of the answer and var-arguments go both ways. Arguments are always by value – both in the call-list as well as in the answer-list. But this "by value-rule" applies for the transport on the serial link only. Locally, the call follows C-conventions (or whatever language you are using). This is best illustrated with the WhoAmI-routine below: Both arguments are pointers that tell the called routine where to store the results. When calling the remote implementation of WhoAmI, the calling record is a structure of type call_body without specific user data (empty). The information contained in the calling record is

sufficient to tell the remote computer what it should do. It will return a reply-body containing a 32bit integer (the version of the remote program) and an XDRstr (see below), a string that will locally be converted to a null terminated string and be stored to where txt points.

2.4.1 WhoAmI

```
int WhoAmI (int* v:out char* txt:out);
```

v version of the remote program

txt pointer to an area that is capable to accept a zero-terminated string of maximum length 50 characters, identifying the remote program.

The calling structure of user data is empty:

```
struct cWhoAmI {void}; // empty used data upon call
```

The returned user data are

```
struct rWhoAmI {
    int v;
    XDRstr txt;
};
```

XDRstr is a 32bit integer specifying the number n of bytes to follow. Then the characters of the string follow and possibly some filler bytes to make the string length a multiple of 4. The number n does not include these filler bytes.

2.4.2 Set ROMP

Configuration parameters of the tracker as well as site and other parameters are stored in a nonvolatile storage area. A working copy of these parameters is maintained in RAM. We thus need three routines: One to upload the parameters to RAM, an other to download these parameters e. g. for editing and finally a routine that reads or writes the parameters from EROM to RAM or vice versa. The EROM is implemented as a linked list in a block of the flash of the AT91SAM7x512 processor.

The type of the ROM-data is of type IROM (for INTRA-ROM) and is defined as follows:

```
typedef struct irom {
    IROMPtr next; // is nil (0) if this irom is valid
    unsigned int Vers; // identifies version of this structure
    float serno; // serial number coded as intra.electronics
    int aofs[2]; // offsets of PA and SA " in encoder counts.
    int range[2][2]; //range[axis,low..high] " in encoder counts
    float gears[2]; // ratios of gears PA and SA. Total ratio
    // motors to axis
    int tcm[2]; // coefficients for target control loop,
    // multiply
    int tcd[2]; // coefficients for target control loop -
    // divide
    int scm[2]; // coefs for speed-control multiply
    int scd[2]; // coefs for speed-control - divide
    float sofs[2]; // offsets [arc] of sun monitor PA and SA
    float Io; // extraterrestrial signal of sun sensor
    float sigma; // coef. of extinction
    float lowelev; // measurements below are jettisoned
    float sunrange[2]; // lower limit of normalized signal and min
    // azimuth- range data must show to be considered
    // valid
    float sunfrac; // analyzed if more data than fraction in
    // limits[2]
    float sun2rad // rwb 100303 converts sunsensor signals to
    // radians. Vers becomes 0x101 now
    int serpa; // coded parameters for UART0 and UART1
};
```

```

float alp[3];           // alignment parameters zenith (zd,az) and
                        // alignment offset of PA. [rad]
float site[3];         // latitude , longitude both in [rad] and
                        // height [m]
                        // N > 0           E > 0
unsigned int tbits;    // bitwise assignment for test-output - see
                        // trackdef.def
int ChkSum;           // twos complement of preceding data
} irom;

```

The serial parameters serpa allow to select the baud rate – all other serial parameters are firm and fix: 8 bit data, 1 stop bit, no parity and no handshake. The baud rate for UART0 is coded in bits (0: 19200, 1: 38400; 2: 57600 and 3: 115200). serpa=0 defaults to 9600. Decoding starts with the lsb and the baud rate according to the first nonzero bit is put into effect. Similarly for UART1 which starts at bit 16 for 19200 etc.

The function to upload the parameter to the trackers RAM is:

```
int SetROMP(IROM rom:in);
```

rom of type IROM is the structure containing all tracker- and site-specific data

The call-user data are

```
struct cSetROMP {
    IROM rom;
}
```

and upon return

```
struct rSetROMP {
    void;
}
```

2.4.3 GetROMP

```
int GetROMP(IROM *rom:out, int romstatus:out);
```

when calling GetROMP, rom must point to a memory area that is able to accommodate the type IROM.

The calling structure cGetROMP is empty. The return-structure includes the ROM-data and a status. This status is the result of GetROMP and is coded as follows:

bit	Description
1	Data valid, but defaults only
2	checksum-error. Data not valid.

```
struct cGetROMP {
    void;
};
```

and the returned data are:

```
struct rGetROMP {
    IROM rom;
    int romstatus; //returned as result of GetROMP
}
```

2.4.4 ROMPrw

```
int ROMPrw(int write:in, int error:out);
```

if write is nonzero (true), the working copy of type IROM is written from RAM to ROM – otherwise the direction is from ROM to RAM. A zero is returned in error if the operation was successful, otherwise the result of the function is 1.

```
struct crROMPrw { // identical for call and return
    int write_error; // direction upon call, result of operation upon return
}
```

2.4.5 SetDateTime

```
int SetDateTime(datetime_t datetime:in);
where
```

```
typedef struct datetime_t {
    int yy, mm, dd; // year e. g. 2009, month and day
    int hh, mi, sec; //hour, minutes and seconds
    int dow; // day of the week (not used by firmware)
} datetime_t;
```

The routine sets the RTC according to the arguments specified. Note that the year should be specified as a four digit number, because otherwise, the century-flag of the RTC can not be handled correctly.

```
struct cSetDateTime {
    datetime_t datetime;
}
```

The routine returns nothing, hence rSetDateTime is empty.

2.4.6 GetDateTime

```
int GetDateTime(datetime_t *datetime:out)
```

The call-structure is empty and for the return data structure, the type cSetDateTime shall be assumed.

2.4.7 SetMode

```
int SetMode(mode_t mode:in, int *err:out)
```

The type Tmode is an enumeration (INIT, SUN, CLOCK, REMOTE, TEST).

If mode > REMOTE an err of 1 is returned – otherwise err of 0 is returned.

Two remarks are in order:

- When the current mode is INIT, the user should firstly verify that the current position is valid prior to commanding any other mode. Failure to do so could result in damages to the tracker, because the axis limit might not work as intended.
- The TEST-mode should not be commanded externally. It is set internally (by the tracker) during special operations. An attempt to set the mode to TEST using SetMode is ignored and err = 1 is returned.

The call- and return data structures are the same – a 32bit-word goes up (mode) and a 32-bit word comes back (err).

2.4.8 GetMode

```
void GetMode( mode_t *mode:out, subm_t *submode:out);
```

The type subm_t is an enumeration (DAY, EVENING, WAIT24, WAITZERO, REWIND, MORNING).

The call-data structure is empty. The reply data are

```
struct rGetMode {
    mode_t mode;        // coded into 32bit word
    subm_t submode;    // 32bit word
}
```

2.4.9 SetPos

This routine should be called in mode REMOTE. If called in INIT, it has no effect. If called in SUN or CLOCK mode, its effect will be overridden by the program within a rather short period of a few seconds at maximum. (Note: It is for safty reason that this routine does not force REMOTE-mode but leaves current mode untouched).

SetPos sets the target-angles accordingly. The angles are specified either in the astronomical system or in the tracker system – depending on the value of the first parameter.

```
int SetPos(cosys_t cosys:in, float p1:in, float p2:in, int *err:out)
```

```
typedef enum {ASTRO, TRACKER} cosys_t
```

SetPos interprets a cosys of 0 as ASTRO and any nonzero cosys as TRACKER.

p1 is the azimuth or primary axis angle, p2 the elevation or secondary axis setting. Angles are specified in radians.

If a position outside the allowed range of an axis is commanded, the target is set accordingly, but the corresponding axis will stop a its limit of range.

```
struct cSetPos {
    Tcosys cosys;
    float p1, p2;
}
```

```
struct rSetPos {
    int err;        // result of function
}
```

2.4.10 GetPos

```
void GetPos(mode_t *mode, subm_t *subm,
    polarf_t *astro_trg:out, // target-az,el astro-system [rad]
    polarf_t *track_trg:out, // target-PA,SA tracker-system [rad]
    polarf_t *astro_cur:out, // current az,el astro-system [rad]
    polarf_t *track_cur:out, // current PA,SA tracker-system [rad]
    polari_t *ecounts:out,   // current encoder counts PA,SA [cnts]
    polari_t *hcounts:out)  // current hall-counts PA,SA [cnts]
```

returns mode and submode, target- and current positions in various systems/units.

The type polarf_t is a two-element vector of float elements (see below) and polari_t is a two-element vector of int elements (Both float and int are 32 bits, float is according to IEEE).

```
typedef struct polarf_t {
    float v[2];
} polarf_t;
```

The call structure is empty. The response-structure contains 8 float and 4 integers starting with astro_trg.v[0] and ending with hcounts.v[1];

2.4.11 GetSun

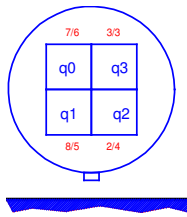


Figure 1
Front view of the quadrants of the sun detector. The numbers indicated are pins of the sensor/connector resp.

```
void GetSun(int *q:out);
```

return intensity-values of the 4 quadrants of the tracker. Units are ADC-readings, normalized with the ADCs full scale (hence something from 0..1).

The call-structure is empty.

```
struct rGetSun {
    float q[4];
}
```

2.4.12 GetMem

```
int GetMem(WORD32 adr:in, int n:in, char *bytes:out);
```

GetMem returns n bytes from memory, starting at adr. n must be in the range [0..128]. If it exceeds these limits, it is cut down accordingly.

```
struct cGetMem {
    int adr;
    int n;
}

struct rGetMem {
    char bytes[n]; // n data-bytes
    char filler<>; // 0 to 3 filler bytes to get a multiple of 4 bytes
}
```

2.4.13 SetMem

```
int SetMem(WORD32 adr:var, int n:var, WORD32 bytes:var);
```

SetMem writes (n=1,2 or 4) bytes to the memory, starting at address adr. Independent of n = 1,2 or 4, bytes is always transmitted as a full 32-bit word – but the 8 lsb, 16 lsb or all 32 bits are valid resp. Unused bits are cleared.

Upon return and if invoked with positive n, adr contains the address that it has written to, n contains an errorflag (0 if ok) and bytes contains the value written to adr. The call and response structures are thus the identical:

```
struct c/rSetMem {
    WORD32 adr;
    int n;
    WORD32 bytes;
}
```

A special mode supporting tests is available when n < 0. The argument adr upon call for n < 0 is irrelevant and depending on the value of n, bytes is written to a predefined variable. Upon return, adr then contains the address of this variable, n an error-flag and bytes the (unmodified) value written to the variable. Currently only one such test variable is defined for

n = -1: Heatertest

The value in bytes is written to a variable "testgrad" (value is in degrees C). This alters the flow of the periodic heater-control-task. If it finds testgrad != 0 it uses² it instead of the temperature value read from the ADC to control the PWM of the heater.

2.4.14 FindZero

```
int FindZero(int search:in, int error:out);
```

start search of zero marks. Bits in search are used to signal which axis and in which direction the search should be started. The coding of the bits is based on the enumeration type tflags

```
typedef enum tflags {
    PAccwsearch = 1,           // ccw is the standard way
    PAcwsearch = 2,           // will search for a negative edge of the z-
                               // signal
    PAzeronotfound = 4,       // moved to end of move-interval
    PAzerofound = 8,          // success - the flag was seen
    PAhe_mismatch = 0x10,     // hall/encoder mismatch
    APovalid = 0x20,          // zero-operation was successful or nv-
                               // values ok
    // SA uses the same flags - just rotated left 8 bits.
    SAccwsearch = 0x100,      // and the same for the SA
    SACwsearch = 0x200,
    SAzeronotfound = 0x400,
    SAzerofound = 0x800,
    SAhe_mismatch = 0x1000,
    SApovalid = 0x2000,
    Force32 = 0x80000000} tflags;
```

Clearly, only a set formed with one or a pair of the search-flags PAccwsearch, PAcwsearch and SAccwsearch, SACwsearch makes sense for FindZero. If non of these bits is set, the call is a nop, but will return 1. The routine simply triggers the search, forces mode INIT and returns.

```
struct c/rFindZero {
    int search; //search is replaced by an error-variable upon return
}
```

2.4.15 ChkAxis

```
void ChkAxis(int *status:out);
```

The status returned is bit-coded as a set of tflags – see also FindZero:

The call-structure is empty, return-structure holds the 4-byte quantity status.

2.4.16 GetLog / Clearlog

```
void GetLog(int n:in, char *txt:out);
```

GetLog returns the nth line of the message-log in string. Linecount starts with zero. The cr – and if present in the log – also the lf are included in string. An empty string signifies the end of the log buffer. For efficiency considerations, this routine should be called in a loop where n grows from zero until an empty string is returned.

When calling GetLog with a negative n, the log-buffer is cleared and an empty string is returned.

² testgrad is cleared during program startup.

```
struct cGetLog {
    int n;
}
```

```
struct rGetLog {
    XDRstr txt; // see also WhoAmI for an explanation of XDRstr
}
```

2.4.17 RunMotors

```
int RunMotors(int flag:in, int pamot:in, int samot:in);
```

This routine allows to operate or stop any of the motors. The argument flag is interpreted as a boolean. If it is unequal zero, the mode is set to TEST and the two motors get a dutycycle as specified in pamot and samot. Dutycycles are specified in ppm – including the sign of the motion. Hence pamot and samot should fall in the range [-999999...999999].

If flag is zero, the mode is reset to INIT and both motors are stopped.

The call-structure contains three 32-bit words, flag, pamot and samot. The response-structure is empty.

2.4.18 GetADC

```
int GetADC(tsig sigmode:in, float *sigs:out);
```

GetADC returns the ADC-values of the 8 channels of the 10-bit ADC of the controller. The data are either in raw AD-counts, Volts or converted to physical signals – according to the channel. The values returned are mean values of a period of 100 ms (50Hz) or 95.24 ms (63 Hz). The routine returns the currently valid readings (which are updated at a rate of 100ms or 95.24 ms resp.). The full scale of the ADC corresponds to 3.3 V.

```
typedef enum tsig {eRAW, eVOLT, ePHYS} tsig; // mode of signals from adc
```

channel assignments are:

ch	Signal	Units	Description
0:	UPWR	VDC	DC power (24 VDCnom) from a voltage divider
1:	UTEMP	°C	KTY13-6-based temperature sensor of board temperature
2:	UCUR0	mA	current from base-shunt of motor0-driver
3:	UCUR1	mA	current from base-shunt of motor1-driver
4:	q0	V	Signal from sun-sensor quadrant 0
5:	q1	V	Signal from sun-sensor quadrant 1
6:	q2	V	Signal from sun-sensor quadrant 2
7:	q3	V	Signal from sun-sensor quadrant 3

The call-structure of GetADC contains the 32-bit quantity sigmode. The response-structure contains an 8-eleemt array of float-types sigs[0]..sigs[7].

2.4.19 SetLogMode

```
int SetLogMode(tlogm lognew:in, tlogm *logwas:out);
```

SetLogMode sets the loglevel specified in lognew and return the previous log-level in logwas.

```
typedef enum tlogm {SEVERE, SHORT, EXTENSIVE} tlogm;
```

The call- and response structure are identical: In both cases a single 32-bit variable - lognew upon call and logwas upon return.

3 Glossary

[rad]	arc units (radians) – full rotation is 2π
[°]	degrees – full rotation is 360°
[cnts]	dimensionless quantity – typically a counter
PA	Primary Axis (vertical axis of tracker) – loosely corresponding (depending on the quality of the initial alignment during installation) to the azimuth axis.
SA	Secondary Axis (horizontal axis of tracker – moving with the PA) - loosely corresponding (depending on the quality of the initial alignment during installation) to the elevation axis

- end of text -