1 General

1.1 Revisions

1.2 Abbreviations/Definitions

byte - 8 data bits (= one octet)

D_Iface - The interface used with this protocol.

D_Unit - Any unit being connected to, and supervised and/or controlled by the D_Iface.

D_UnitObj - Any D_Unit internal object, which is being supervised and/or controlled via the here described protocol. Examples are: push-buttons, LEDs, IDstring and measured/controlled numerical parameters.

D_UnitAdr - The physical address of a D_Unit. Its value may be fix, depending on subrack PIU slot or reconfigurable. Product documentation for the D_Unit must describe this.

1.3 Supported Interconnection Structures

The protocol is intended for a mainly (though not only) master/slave oriented communication. The connection between units is either point to point (Gateway to Star Controller) or a chain connection of units (Chain Controllers), each having two D_Iface interfaces.

1.4 General Features

The protocol has the following characteristic properties:

- Compressed Self Expanding header. This allows the protocol to be flexible for various applications and yet efficient. Most fields of the header are minimum three bits, but expand automatically as their values so require. The minimum header overhead is 5 bytes.
- Use of general standardised objects, D_UnitObj (see definition above). These objects define what types of requests and responses that are valid and the formats of associated data of the message, if any. The objects are generalised to serve more than one purpose. The idea of this concept is to reduce product documentation (by reference to the objects) and enable reuse of software code (as is the idea of protocol itself). Up to 255 D_UnitObj objects, each of 255 object types, can be used in each D_Unit.
- The protocol has provisions for changes and extension by use of a number (as a header field) and a letter (as a controller property).
- Among other address modes, the protocol can use relative addressing - by number of hops - in daisy chain configurations. This allows a very simple address configuration by the cabling.

2 Protocol Description

2.1 General

The protocol is message based. Most messages are master/slave oriented, a master sends a request causing a response message from a slave. This is however no restricition, the protocol also allows any units to exchange messages point to point.

The following general size restrictions are valid for all SVIFT protocol messages:.

Message Parameters

MAX_SM_LEN:	32 bytes maximum allowed SVIFT message size (unframed data).
MIN_SM_LEN:	5 bytes minimum SVIFT message size.

SVIFT protocol messages normally require some form of packaging into “frames”. That framing functionality is however deliberately left of the SVIFT messages to make it useful on several communications media, since some of these media lower layer protocols already have framing provisons.
The most essential functionality required by the SVIFT protocol from a framing layer is to determine message length - the message format described here needs message length, but has no provisions of its own to determine it. Depending on media and application, there might also be other requirements: how to separate from other protocols, tagging, and so on.

2.2 Special Data Types

For compression/expansion of various information, the SVIFT protocol uses expansion bits to tell the size of data fields. Two such uses have been assigned special data types, and are described separately in the two following subchapters for shorter notation in the rest of the document: the EBYTE data type and the DENIB data type.

2.3 The EBYTE Data Type

The size of an EBYTE (Expanding BYTE) is minimum one byte (byte 1 of the illustration below), and if so the lower significance 7 bits (A0,...A6) are used for binary number representation of a parameter A. If the value of A exceeds what can be represented by 7 bits (i.e. unsigned value above 127), the A_E1 expansion indicator bit is set (=1) and another byte is inserted immediately after the original byte. The new byte represents 6 more bits (A7,...A13) of A and has a new expansion indicator bit A_E2 for further expansion. Writing EBYTE(A) is short for:.

Byte	MSB	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	LSB
1	A_E1	A6	A5	A4	A3	A2	A1	A0
2	A_E2	A13	A12	A11	A10	A9	A8	A7
3	A_E3	A20	A19	A18	A17	A16	A15	A14

where bytes 2 and 3 only are present if so indicated by expansion indicators A_E1 and A_E2 being set. Byte numbering indicate the order in which the bytes are sent over the interface.

The insertion of new bytes only depends on the value of A; if its value is low one single byte might be enough, but it can be expanded to any size required. When no more bytes are required, the expansion indicator bit of the last byte is cleared (=0).

2.3.1 The DENIB data type

The minimum size of a DENIB (Double Expanding NIBble) is also one byte. It is however used for representation of two values. Writing DINIB(A:B) is short for:

Byte	MSB	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	LSB
1	A_E1	A2	A1	A0	B_E1	B2	B1	B0
2	EBYTE(A9,..A3)							
3	EBYTE(B9,..B3)							

where A0,A1,A2 are the three lower order bits of A (A0 identifying least significant bit, higher numbers indicating more significant bits) and B0,B1,B2 have the corresponding meaning for the bits of B. Byte numbers 2 and 3 are extension bytes, that are only present if the values of A and B can not be represented by the 3 bits of byte 1 (i.e. their binary value exceeds 7). The extension bytes are inserted individually for A and B and in the order shown above if both are present. The presence of a extension byte for A and/or B is indicated by setting individual flag bits, A_E1 (A extended) and B_E1 (B is extended). If any of these bits are zero, the corresponding extension byte is not present. Byte numbering indicate the order in which the bytes are sent over the interface.

Since each of the extension bytes are EBYTE data types, they may be further expanded to any value if so required.

2.3.2 The STRING data type

The string data type represents a series of bytes, terminated by a null (binary value 0) byte. Though it can be used for transfer of any data, it is typically used for transfer of ASCII text, such as names. The terminating null byte is to be considered included in the string. Writing STRING(NAME) is short for any string that is further referred to as variable NAME. This definition is identical with use of strings in the C programming language.

2.4 SVIFT Message Format

The format of a SVIFT message is as follows:

Data	Presence	Description
DENIB(HFLG:HPNR)	Mandatory	HFLG: Header Flags. Descriptor of the header format. Indicates which of the optional fields are present. HPNR:Header Protocol number. Used for future expansions of the protocol. For all SVIFT version so far, this value is always = 1.
DENIB(DMOD:DADR)	Mandatory	DMOD: Destination address mode. DADR: Destination Address.
DENIB(SMOD:SADR)	Mandatory	SMOD: Source address Mode. SADR: Source Address.
OTYP (one byte)	Mandatory	OTYP: Determines which Object Type this command/request/response refers to.
DENIB(ONBR:CODE)	Mandatory	ONBR: Object Number. Distinguishes between more than one object of the same OTYP type. Numbering is always contiguous, starting at zero. CODE: Used as command code for requests as well response code.
EBYTE(SQNR)	Optional	SQNR: Sequence number of a message.
Data Field	Determined by CODE and OTYP	The contents of the data field is separately described in chapter “Data Field”
ECHK (one byte)	Optional	Extra Checksum. Sum of all bytes of the SVIFT message, except the ECHK byte itself, truncated to 8 bits.

where data is sent in order top to bottom. All bytes down to the Data Field are also commonly referred to as the SVIFT message “header”.

The values HPNR and the presence of optional header fields, as determined by HFLG, is always kept unchanged from requests to responses. The only exception from this is some of the error messages.

In trying to handle a message, the fields shall be examined in the following order: HPNR, HFLG, ECHK (if used), DMOD, DADR, SMOD, SADR, OTYP, ONBR, CODE and last the SQNR (if used) field. This as a further guidance to interpretation of possible error response messages.

If the value of any of the fields of a message exceeds the maximum value supported by a controller, the controller must assume that the message is destined to another controller.

2.4.0.1 HFLG fields

The HFLG field is used for indication of which optional fields are present in the header:

Hexa decimal value	Name	Description
0x01	HFLG_SQNR	If this bit is set, a SQNR field is present in the header, else not. The use is optional (to simplify handling of multiple concurrent requests within the issuer).
0x02	HFLG_ECHK	If this bit is set, a ECHK field is present immediately after the message data, else not. The presence of the ECHK field forces the receiver to test it for validity. The use is optional.
0x04	HFLG_REQU	If this bit is set, the message is a request. Otherwise it is a response.

The value of HFLG is determined by the issuer of a message. Except for the HFLG_REQU bit, possible response messages must have exactly the same bits set (and the corresponding fields of the header present).

2.4.0.2 HPNR field

The only currently allowed value in this field is 1. The field is intended for future extensions of the protocol. Since revision C of the protocol, all chain controllers must also be able to pass messages on without complaints even if the HPNR is 2 (but not respond to them).

2.4.0.3 DMOD, DADR, SMOD and SADR fields

The destination and source D_UnitAdr's of a message. The entire DENIB(DMOD:DADR) and DENIB(SMOD:SADR) fields swapped in a possible response. (Exception to this is broadcast messages, see "Addressing and general behaviour below"). Special values for DMOD, SMOD are:

Hexa decimal value	Name	Description
0x00	AMOD_PHYS	Physical Address Mode. Address matches if DADR = D_UnitAdr. The message is terminated and responded to by the unit having an address match.
0x01	AMOD_BCST	Broadcast Address Mode. Address matches all units. The message is responded to by all units, and never terminated. The corresponding address field value must be zero. May never be used as SMOD. Possible responses always use SMOD = AMOD_PHYS.

Hexa decimal value	Name	Description
0x02	AMOD_RELP	Relative Physical Address Mode. When used as DMOD, the DADR is first decremented, if it then becomes zero, the address matches, and the unit will terminate and then respond to the message. When used as SMOD, the SADR value is incremented. The incrementation or decrementation shall take place before the contents of the message is further evaluated - it thus affects passed on as well as responded to messages.
0x03	AMOD_RELB	Relative Broadcast Mode. When used as DMOD, the DADR value is first decremented, if it then becomes zero the message is terminated. The message is responded to by all units. May never be used as SMOD. Possible responses use SMOD = AMOD_RELP.

2.4.0.4 CODE field

This is the request or response code of a message. The following values are defined:

Hexa decimal value	Name	Description
Request/Response type (request is even, response is odd).		
0x00	CODE_Read	Read a D_UnitObj. Always causes a response message to be sent (CODE_Read or CODE_Err).
0x01	CODE_Write	Write to a D_UnitObj. Always causes a response message to be sent (CODE_Write or CODE_Err).
0x02	CODE_Start	Start or enable a D_UnitObj function.
0x03	CODE_Stop	Stop or disable a D_UnitObj function
0x04		
0x05		
0x06	CODE_Name	Get the name of a D_UnitObj. Always causes a response to be sent (CODE_Name or CODE_Err)
0x07	CODE_Err	Error. If the message is a response, it indicates that a request was not carried out due to errors. The contents of this message indicates the kind of error.
0x08	CODE_Echo	Echo back the message data field
0x09	CODE_Info	The message is for information.
0x0a	CODE_Clear	Clear a D_UnitObj function. The exact meaning of this command is described together with each D_UnitObj object type.

The definition of each D_UnitObj describes what CODE field values can be used, and possibly also defines its function in more detail.

2.4.0.5 OTYP field

This is the D_UnitObj type of object that the CODE request or response refers to. Defined OTYP values are listed together with their associated CODE values and data fields in chapter “D_UnitObj types” below.

2.4.0.6 ONBR field

This field identifies which, of possibly many, objects of the same OTYP object type that the CODE request/response code refers to. Objects are always numbered contiguously (without left out numbers) from zero upwards. Thus, if only one object of a certain OTYP type exist in the equipment, its ONBR number is zero.

2.4.1 Addressing and General Behaviour of Equipment

Star controller
A star controller, acts upon a message if DADR matches any of the addresses of the connected PIUs. If the destination address does not match, the message is dropped without errors. A broadcast message will affect (and cause possible responses for) all connected PIUs. Possible responses are sent back via the normal interface.

Chain Controller
In this type of equipment, if DADR does not match, the message is passed on via the other interface (i.e. not the interface where the message arrived). Possible responses are always sent back to the interface where the request message arrived. This means a message will be forwarded to the correct PIU via a number of other PIUs and a possible response will be sent back via the same path of PIUs in the opposite direction. A broadcast message is as well parsed as passed on. Messages sent to unused addresses, and also broadcast addressed messages, will thus reach the other end of the daisy-chain of PIUs and the there (possibly) connected gateway.
The use of the SADR field of the protocol header allows any PIU to communicate to any other PIU without gateway assistance.
A chain controller is capable of handling relative address modes.

Gateway
A gateway only parses messages that is expects as response to sent out requests, and the possible error messages that may appear instead of responses. This means a gateway is insensitive to messages that reach it by mistake (due to bad addressing).

2.4.2 Data Field

The format of the data field for most messages is described together with the description of the object types, see chapter “Object Types” for further information. This chapter therefore only describes general message types that are not referred to any object type in particular.

2.4.3 General Messages

2.4.3.1 Error Message

Error messages are sent as response to incoming requests in case of errors. The error message always returns the same header HFLG, HPNR, OTYP and ONBR values as the request. Exceptions to this rule are:

- Extra checksum is never used in error messages. If the HFLG_ECHK bit was set in the request that caused the error message, it is cleared in the response (and the EBYTE(SQNR) field is removed).
- The Err_BadHflg and Err_BadHpnr errors also cause header information to change, see description below.

The error message data field format is:

Data (one byte each)	Description
RCODE	The header CODE value of the message that caused the error/failure.
ERRNR	A pre-defined value according to the table below. (NOTE: ERRNR deals with message format errors, not to mix with ERRNO which deals with controller functional failures.)

The following ERRNR error codes are have general use, they may appear with any object type. Testing for these errors is mandatory. Testing for errors Err_BadHflg and Err_BadHpnr is performed on all messages, while testing for other error numbers only is performed by the addressee :

Hexa decimal value	Name	Description
0x00	Err_BadHflg	The equipment does not support this HFLG value. This error message is always given with HFLG = HPNR = OTYP = ONBR = 0, and the incoming HFLG value as data field RCODE value.
0x01	Err_BadHpnr	The equipment does not support this HPNR value. This error message is always sent with HFLG = HPNR = OTYP = ONBR = 0, and the incoming HPNR value as data field RCODE value.
0x03	Err_BadEchk	The extra checksum ECHK was incorrect
0x10	Err_BadObjType	The addressed object type is not used in this kind of equipment, or does not exist.
0x11	Err_BadObjNr	The object number is out of range for the addressed type of equipment.
0x12	Err_BadCode	The CODE value is not valid for this object type.

Hexa decimal value	Name	Description
0x20	Err_BadData	The size of the data field, as calculated from W_LENGTH minus header size, is not correct for this OTYP object type and CODE request code.
0x21	Err_BadResp	The request would cause a response that exceeds the maximum allowed by the protocol.

The following ERRNR error codes can only appear with a certain object type if so mentioned in its description, which may also further define its meaning:

Hexa decimal value	Name	Description
0x30	Err_BadRange	Any of the Data Field parameters is out of range
0x31	Err_ReqFail	The request was not successfully carried out.
0x32	Err_Blocked	The requested functionality is blocked by a protection mecahnism
0x33	Err_Param	One of the parameters is wrong from some respect (also see Err_BadRange). Example of use is bad password.

2.4.3.2 Name Messages

All D_UnitObj objects have names, which can be read by the CODE_Name messages. Names are represented by ASCII strings of length 0 - DMAXNAMLEN plus a terminating null byte. DMAXNAMLEN is equal to 16 bytes for the current version of the protocol.

The D_UnitObj objects that use individual bits (OTYP_ROFLB, OTYP_EVFLB) and states (OTYP_4STCTL, OTYP_NSTCTL) also have names for each individual bit and state. CODE_Info is used to read these names.

For message formats, see the description of each object type in chapter 3.

Routines that use names in order to recognise objects should always convert names to all upper case letters and remove blanks (hex 0x20) before comparison to the expected string. It is also recommended not to use blanks in names.

2.5 Alarms

The SVIFT protocol supports alarms to be handled by use of OTYP_ROFLB and OTYP_EVFLB objects. Conditions that mean problems in most applications of the D_Unit should use these objects types and have the corresponding AMASK and BMASK values designed according to the classification in chapter 2.5.1. The instance and bit names should be chosen to give the best possible guidance to any operator of the nature of the problem. Specific applications of the D_Unit may override the AMASK and BMASK values or even supervise other objects types if so required.

If OTYP_GROUP objects are used, OTYP_ROFLB and OTYP_EVFLB objects contained in these GROUPs must have their alarm bits duplicated also to a bit of a central (non-group contained) ROFLB or EVFLB. In other words, a supervisor should not need to read a GROUP structure to find any possible alarms - any bits in these GROUP contained objects should only be used for finer resolution of the problem. The most obvious reason for this rule is backward compatibility, supervision equipment designed from revision A or B of this specification have no knowledge of OTYP_GROUP. GROUP contained ROFLBs or EVFLBs may still use non-zero AMASK and BMASK values to provide further guidance the origin of the problem.

2.5.1 Alarm Classification

Flag oriented objects also contain information on recommendations whether the flag should be considered an alarm or not. There also two different alarm priorities defined. These general meaning of these alarm priorities are as follows:

A - alarm: Failures of this category require immediate attention.

B - alarm: Failures of this category can be corrected at the next regular service.

Flags belonging to neither the A or B category are for information only. They may be used freely for other purposes.

This classification is recommendation only, since severity of failures may depend on the application. It should be described by the product documentation, so the user can determine whether his classification agrees, or if overrides have to be made.

2.6 Standard Attributes

The following objects are recommended to be implemented by all SVIFT products:

Attribute	Recommendation
ProductIndividualData	Should be implemented as OTYP_NVSTR with: ONBR=0, NAME=“ProdIndivData” (exactly spelled). TOTSIZ=100 The first byte of the string determines the length of the ProductIndividualData read by other protocols, bytes following contain the actual ProductIndividualData string.
Front Panel “MIA” LED	Should be implemented as OTYP_4STCTL with: ONBR=0 NAME=”StdLED” state names: 0=”Off”, 1=”SlowFlash”, 2=”FasFlash”, 3=”On”

Attribute	Recommendation
Front Panel Push Button (if present)	Should be implemented as OTYP_EVFLB with: ONBR=0 bit MASK=1 bit NAME=”StdPB” The instance name of the EVFLB may be freely chosen.

2.7 Controllers

2.7.1 General Behaviour of D_Iface Controllers

A message is acted upon by the receiving equipment if its destination address (header DADR field) matches its D_UnitAdr. The contents of the DADR and SADR fields are swapped in a response (so DADR is the destination, and SADR is the source of the response).
Address value DADR_BCST (meaning broadcast message) is never used in a response, it is instead replaced by D_UnitAdr of the responding unit

A Gateway sends out the request messages it needs to fulfil its function as a bridge to other interfaces (the Maintenance Channel Level1), and only cares about the incoming messages that are relevant responses to those. The only exception to this general behaviour is that it also responds to messages concerning its OTYP_CONTR D_UnitObj. The gateway never issues any Error Messages concerning response messages or messages concerning other than its OTYP_CONTR D_UnitObj.

A Star Controller responds to requests to the D_UnitAdr address range that are present in a full configuration (normally full configured subrack) and broadcast messages. If a PIU is not mounted, it responds using the proper Error Message. Requests to DADR values that are never present (even in a full configuration) are discarded.

A Chain Controller only responds to the D_UnitAdr address of the PIU it is itself located on, and the broadcast address. For other values of the DADR field of incoming messages, it passes the message on to its other D_Iface (i.e. not the D_Iface where the message arrived). Responses are sent back to the same interface where the message arrived, and are not passed on. Broadcast messages are as well responded to as passed on.

Using these transfer mechanisms, the Gateway sees very little difference if the D_Iface connected equipment uses a Star Controller or a Chain Controller.

3 D_UnitObj Types

The following subchapters are detailed descriptions of the defined D_UnitObj types and their associated CODE values and data fields. The following table shows the OTYP numbers assigned to the types and gives an overview of the functionality:

MOTY value (HEX)	Name	Description
0x00	OTYP_CONTR	The D_Iface controller itself. This object handles parameters that are common to all object types or no object type in particular.
0x01	OTYP_NVSTR	A Read/Write non-volatile string of bytes.
0x02	OTYP_EVFLB	A byte of event flags.
0x03	OTYP_ROFLB	A read only flag byte.
0x04	OTYP_4STCTL	A four state control.
0x05	OTYP_8ROSAN	A read only 8-bit signed analogue value, using scale factor
0x06	OTYP_8ROSBN	A read only 8-bit signed binary value
0x07	OTYP_NSTCTL	A general state control (any number of states).
0x08	OTYP_GROUP	A grouping of other objects
0x09	OTYP_OUTB	A byte of output bits.
0x7f	OTYP_CONF	A configuration object.
0x80	OTYP_FINFO1	A fan controller general info object.
0x81	OTYP_FINFO2	A fan controller object for general override

3.1 Object OTYP_CONTR

This D_IfaceObj represents the controller of the D_Iface itself. Its implementation is mandatory but its use is optional. It may be used by the master to optimize its strategy, and to test the D_Iface.

The object supports the following header CODE values:

CODE_Read	
Function: Read Controller General Parameters	
Data (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
TYPE	TYPE:The controller type 0=Supervisor, 1 = Chain Controller
PREV	PREV: The ASCII representation highest revision of this protocol (revsion of this document) that the controller supports. Since revision D is the currenlty latest revision, PREV = 0x44 for new designs (but may also be 0x41,0x42 or 0x43 for older designs.
ERRNO	The last internal error type that occurred within the controller. (NOTE: ERRNO deals with controller functional failures, do not mix with ERRNR that deals with message signalling errors.) For recommendations on ERRNO coding, see chapter 3.1.1
SEQ	The sequence number of the ERRNO. It is incremented each time the controller updates ERRNO and can therefore be used to determine whether more errors of the same type are occurring. SEQ is never initialized, only incremented by the controller software. The value at power up of the controller is undefined (hardware dependent).
EPROT (optional)	A optional sequence of pairs of EPROT, EPREV, specifying other protocol levels (header HPNR values) and highest revisions (similar to PREV above) that the controller supports. EPROT is a binary value, EPREV is a ASCII letter.
EPREV (optional)	
....	

CODE_Info	
Function: Read Controller Functionality	
Data(one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
NUM	A repeated sequence of byte pairs NUM, OTYP that the controller will respond to, where OTYP is the defined object type and NUM is the number of such objects. The last pair of bytes has NUM = 0 and any OTYP value.
OTYP	
...	

CODE_Echo	
Function: Echo data field	
Data	Description
Request:	
(any data)	The number of bytes is determined from the message size.
Response:	
(same as request)	

CODE_Name	
Function: Read CONTR instance name	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
STRING(INSTNAME)	The name of this CONTR object instance.

3.1.1 Recommended ERRNO values

The following codes are recommended for ERRNO at various failures. There is no requirement that all these must be implemented. It is also allowed to add new codes as needed. If they are of more general nature, please notify the author so they can possibly be added as recommendations in future revisions of this document.

Name	Value (hex)	Description
Startup/Reset		
EPWRON	0x00	A normal startup (reset). If power up can be detected, it was the reason of the startup.
EFRESET	0x01	A failure caused a reset/restart. The type of failure could be any that is detected by the controller, for example: watchdog timeout, illegal opcode trap.
Heap oriented failures		
EBADHEAP	0x10	Heap was corrupted (but reinitialised).
EBADFREE	0x11	A free() call using invald pointer was issued.
ENOHEAP	0x12	A malloc (or similar) call was not successful due to heap limitations.
Signalling related failures:		
EOSIGOVF	0x20	Output overflow. This situation may occur when a master issues requests, that cause long response messages, too rapidly. The problem might be either total output buffer size or limitations in the number of outgoing messages.

3.1.1.1 Object OTYP_NVSTR

The NVSTR object type represents a size TOTSIZ non-volatile string, typically stored in EEPROM, which can be read and written. The total size of the string may for example be determined by issuing a CODE_Read with STARTP = NUM = 0, which will never fail (see below).

The object supports the following header CODE values:

CODE_Read	
Function: Read OTYP_NVSTR data	
Data (one byte each unless otherwise specified)	Description
Request:	
STARTP	The starting byte position, range 0,...,TOTSIZ-1. Err_BadRange will be returned if STARTPOS is outside this range.
NUM	NUM: The number of bytes to read.
Response:	
TOTSIZ	The total size of NVSTR in bytes
STARTP	The starting byte position of the returned data.
NUM	The number of NVSTR data bytes returned (may be truncated if part of the requested NUM bytes are outside TOTSIZ).
(num bytes of data)	The NVSTR data

CODE_Write	
Function: Write NVSTR data	
Data (one byte each unless otherwise specified)	Description
Request:	
STARTP	The starting byte position, range 0,...,TOTSIZ-1. Err_BadRange will be returned if STARTPOS is outside this range.
NUM	The number of bytes to write. Err_BadRange will be returned and nothing written if part of the data is outside 0,...,TOTSIZ-1.
(NUM bytes of data)	
Response:	
STARTP	The starting byte position of the returned data.
NUM	The number of NVSTR data bytes written.

CODE_Name	
Function: Read NVSTR instance name	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
STRING(INSTNAME)	The name of this NVSTR object instance.

Note that the maximum SVIFT message size, MAX_SM_LEN, also limits NUM to 16..25 depending on header parameter values. It is therefore not recommended to use NUM > 16.

3.1.1.2 Object OTYP_EVFLB

This object represents a byte of event flags, where each bit position represents a event flag. The flag FLAG is cleared (=0) until the associated event occurs, but then gets set (=1). The flag may be read, cleared, disabled (which implies a clear) or enabled. At least one of the bits have this functionality implemented. Unused flag bits always report disabled status. Enabling or disabling of unused bits will cause no error.

No error messages are returned for redundant requests (for example to disable an already disabled OTYP_EVFLAG).

The state at power up of the controller is disabled (STAT = 0) and cleared (FLAG = 0).

The object supports the following CODE values::

CODE_Read	
Function: Read EVFLB flags and status	
Data (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
STAT (one byte)	Status (0=Disabled, 1=Enabled) for each of the flag bits.
FLAG (one byte)	Event flag (0=not occurred, 1=has occurred) for each of the flag bits.
AMASK (one byte)	Mask for what bits in FLAG are recommended to be treated as level A failure alarms (see chapter 2.5).
BMASK (one byte)	Mask for what bits in FLAG are recommended to be treated as level B failure alarms. (see chapter 2.5)

CODE_Start (enable), CODE_Stop (disable), CODE_Clear	
Function: Enable, Disable (implies Clear) or Clear one or more flag bits	
Data (one byte each unless otherwise specified)	Description
Request:	
BITS	Performes Enable, Disable or Clear of the flag bits set in BITS.
Response:	
BITS	Same as BITS of the request, but with non implemented bit positions cleared.

CODE_Name	
Function: Read EVFLB instance name	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
STRING(INSTNAME)	The name of this EVFLB object instance.

CODE_Info	
Function: Read EVFLB bit name(s)	
Data field (one byte each unless otherwise specified)	Description
Request:	
MASK	MASK uses the same bit positions as the flag byte. Each bit set in MASK requests the name of the corresponding bit. This means 0 to eight names can be requested by the same message. It is up to the requesting part not to ask for more names than will fit into one response message.
Response:	
MASK	
STRING(NAME1)	0 to 8 names corresponding to the bits set in MASK.
STRING(NAME2)	
...	

The following form of CODE_name is also supported, but not recommended (use CODE_Info instead):

CODE_Name	
Function: Read EVFLB bit name(s)	
Data field (one byte each unless otherwise specified)	Description
Same request and Response formats as CODE_INfo described above	

3.1.1.3 Object OTYP_ROFLB

This object represents a byte of condition flags, where each bit position represents one flag. The respective flag bit is set (=1) if the condition is true, else cleared (=0). The flag byte may be read only. At least one of the bits have this functionality implemented. Unused flag bits always report cleared status.

The object supports the following CODE values (also see chapter 2.4.3 description on bit names):,.

CODE_Read	
Function: Read ROFLB flags and status	
Data (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
FLAG	Condition flag (0=not true, 1=true) for each of the flag bits.
AMASK	Mask for what bits in FLAG are recommended to be treated as level A failure alarms (see chapter 2.5).
BMASK	Mask for what bits in FLAG are recommended to be treated as level A failure alarms (see chapter 2.5).

CODE_Name	
Function: Read ROFLB instance name	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
STRING(INSTNAME)	The name of this ROFLB object instance.

CODE_Info	
Function: Read ROFLB bit name(s)	
Data field (one byte each unless otherwise specified)	Description
Request:	
MASK	MASK uses the same bit positions as the flag byte. Each bit set in MASK requests the name of the corresponding bit. This means 0 to eight names can be requested by the same message. It is up to the requesting part not to ask for more names than will fit into one response message.
Response:	
MASK	
STRING(NAME1)	0 to 8 names corresponding to the bits set in MASK.
STRING(NAME2)	
...	

The following form of CODE_name is also supported, but not recommended (use CODE_Info instead):

CODE_Name	
Function: Read ROFLB bit name(s)	
Data field (one byte each unless otherwise specified)	Description
Same request and Response formats as CODE_INfo described above	

3.1.1.4 Object OTYP_4STCTL

The 4STCTL object type represents a four state indicator of any kind. What functionality the four states represent must be described by product documentation.

The object supports the following header CODE values:

CODE_Read	
Function: Read 4STCTL state	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
STATE	The 4STIND present state. (0, 1, 2 or 3)

CODE_Write	
Function: Write 4STCTL state	
Data field (one byte each unless otherwise specified)	Description
Request:	
STATE	Requested new state of the 4STIND. (0, 1, 2 or 3). Other values will cause a Err_BadRange.
Response:	
STATE	Same as request

CODE_Name	
Function: Read 4STCTL instance name	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
STRING(INSTNAME)	The name of this 4STCTL object instance.

CODE_Info	
Function: Read 4STCTL state name	
Data field (one byte each unless otherwise specified)	Description
Request:	
STATE	
Response:	
STATE	
STRING(STATENAME)	The name of the state STATE

The following form of CODE_name is also supported, but not recommended (use CODE_Info instead):

CODE_Name	
Function: Read 4STCTL state name	
Data field (one byte each unless otherwise specified)	Description
Request and response formats same as described for CODE_Info above	

The following error responses have special interpretation for this object type:

Err_BadRange - STATE was outside the allowed range (0,...4).

3.1.1.5 Object OTYP_8ROSAN

The 8ROSAN object type represents a read only 8bit signed analog value, using 2’s complement negative value representation. What parameter the value represents must be described by product documentation.
The object supports the following header CODE values:

CODE_Read	
Function: Read 8ROSAN value	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
VALUE	The value (signed integer)
MULT	VALUE must be multiplied by this value before use (unsigned integer).
DIVI	VALUE must be divided by this value before use (unsigned integer).
EXP	VALUE must be multiplied by 10^{EXP} before use (signed integer).
TYPE	The type of value: 1= Voltage in Volts 2= Current in Amperes 3= Temperature in Centigrades

CODE_Name	
Function: Read 8ROSAN instance name	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
STRING(INSTNAME)	The name of this 8ROSAN object instance.

3.1.1.6 Object OTYP_8ROSBN

This object represents a binary value, that may be read only.

The object supports the following CODE values:,.

CODE_Read	
Function: Read 8ROSBN value	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
VALUE	The value, represented as a 8-bit signed (2's complement if negative) binary number

CODE_Name	
Function: Read 8ROSBN instance name	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
STRING(INSTNAME)	The name of this 8ROSBN object instance.

3.1.1.7 Object OTYP_NSTCTL

The NSTCTL object type represents a state indicator/control of any kind. The number of states is defined by each instance separately. What functionality the various states represent must be described by product documentation.

The object supports the following header CODE values:

CODE_Read	
Function: Read NSTCTL state	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
NUMSTATES	The number of states
STATE	The current state (0,..,(NUMSTATES-1))

CODE_Write	
Function: Write NSTCTL state	
Data field (one byte each unless otherwise specified)	Description
Request:	
STATE	Requested new state. (0,..,(NUMSTATES-1)).
Response:	
STATE	Same as request

CODE_Name	
Function: Read NSTCTL instance name	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
STRING(INSTNAME)	The name of this NSTCTL object instance.

CODE_Info	
Function: Read NSTCTL state name	
Data field (one byte each unless otherwise specified)	Description
Request:	
STATE	
Response:	
STATE	
STRING(STATENAME)	The name of the state STATE

The following form of CODE_name is also supported, but not recommended (use CODE_Info instead):

CODE_Name	
Function: Read NSTCTL state name	
Data field (one byte each unless otherwise specified)	Description
Request and response formats same as described for CODE_Info above	

The following error responses have special interpretation for this object type:

Err_BadRange - STATE was outside the allowed range (0,...(NUM-STATES-1)).

3.1.1.8 Object OTYP_GROUP

The GROUP object type does not represent any functionality by its own, but is rather to be considered a named container of other objects. It is typically used to represent any (repeated) sub-function within the D_Unit. GROUP objects can be contained in other GROUP objects, thus creating a multi level nested hierarchy limited only by message size. Object numbering is started from zero in each GROUP.

For handling of alarms from objects contained in GROUPs, see recommendations in chapter XXX.

The OTYP_GROUP object itself responds to the following two commands only, providing no data is present in the request::

CODE_Name	
Function: Read GROUP instance name	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
STRING(INSTNAME)	The name of this GROUP instance.

CODE_Info	
Function: Read GROUP instance name	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
NUM	A repeated sequence of byte pairs NUM, OTYP that the controller will respond to, where OTYP is the defined object type and NUM is the number of such objects. The last pair of bytes has NUM = 0 and any OTYP value.
OTYP	
...	

CODE_Start	
Function: Pass request/response message to/from a contained object	
Data field (one byte each unless otherwise specified)	Description
Request:	
EBYTE(S_OTYP)	The OTYP of the contained object
DENIB(S_ONBR:S_CODE)	The ONBR of the contained object and the CODE to pass on to the OTYP, ONBR object.
data	The DATA field to pass on to the contained object, format according specifications for that object type. (length >= 0)
Response:	
EBYTE(OTYP)	Copied from request
DENIB(ONBR:CODE)	Copied from request.
data	The response DATA field returned from to the contained object, format according specifications for that object type (length >= 0).

The following error responses have special interpretation for this object type:

Err_BadData - The DATA field could not be interpreted as an EBYTE(), DENIB() combination (any CODE value), and was not zero length with a CODE_Name or CODE_Info request.

Err_BadObjType - The GROUP does not contain any S_OTYP objects.

Err_BadObjNr - The GROUP does not contain any S_OTYP object having number S_ONBR.

3.1.1.9 Object OTYP_OUTB

This object represents a byte of controllable bits. Typical application is ON/OFF control of functionality or digital output pins. Functionality and naming of individual bits should be chosen so that a 1 represents ON or true condition, and a 0 represents OFF or false. The flag byte may be written and read. All bits are cleared at start-up. At least one of the bits have this functionality implemented. Unused flag bits always report cleared status.

The object supports the following CODE values (also see chapter 2.4.3 description on bit names):,.

CODE_Read	
Function: Read all OUTB bits	
Data (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
BITS	The present state of all bits.
MASK	Mask for what bits in BITS are actually implemented/ used.

CODE_Start	
Function: Set (to 1) selected OUTB bits	
Data (one byte each unless otherwise specified)	Description
Request:	
MASK	Mask for what bits are to be set.
Response:	
MASK	Same value as request.

CODE_Stop	
Function: Clear (to 0) selected OUTB bits	
Data (one byte each unless otherwise specified)	Description
Request:	
MASK	Mask for what bits are to be cleared.
Response:	
MASK	Same value as request.

CODE_Name	
Function: Read ROFLB instance name	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
STRING(INSTNAME)	The name of this OUTB object instance.

CODE_Info	
Function: Read ROFLB bit name(s)	
Data field (one byte each unless otherwise specified)	Description
Request:	
MASK	MASK uses the same bit positions as the flag byte. Each bit set in MASK requests the name of the corresponding bit. This means 0 to eight names can be requested by the same message. It is up to the requesting part not to ask for more names than will fit into one response message.
Response:	
MASK	Same as request, but unimplemented bits have been removed.
STRING(NAME1)	0 to 8 names corresponding to the bits set in MASK.
STRING(NAME2)	
...	

3.1.1.10 Object OTYP_CONF

CONF object is similar to the NVSTR object type in that it represents a series of bytes that can be read and written non-volatile memory. Objects of CONF type should however not be announced by the controller (in its response to CODE_Info). This since they are not to be considered “run time” - any changes made to the CONF object are not guaranteed to take full action until after the next reset of the unit.
I addition to the NVSTR object type, CONF also has a protection mechanism by use of password. This functionality is handled by the CODE_Start, CODE_Stop and CODE_Info command codes.
The format/usage of the contents of the CONF object is completely defined by each product.

The object supports the following header CODE values:

CODE_Read	
Function: Read OTYP_CONF data	
Data (one byte each unless otherwise specified)	Description
Request:	
STARTP	The starting byte position, range 0,...,TOTSIZ-1. Err_BadRange will be returned if STARTPOS is outside this range.
NUM	NUM: The number of bytes to read.
Response:	
TOTSIZ	The total size of CONF in bytes
STARTP	The starting byte position of the returned data.
NUM	The number of CONF data bytes returned (may be truncated if part of the requested NUM bytes are outside TOTSIZ).
(num bytes of data)	The CONF data

CODE_Write	
Function: Write CONF data - see also "Note:" below	
Data (one byte each unless otherwise specified)	Description
Request:	
STARTP	The starting byte position, range 0,...,TOTSIZ-1. Err_BadRange will be returned if STARTPOS is outside this range.
NUM	The number of bytes to write. Err_BadRange will be returned and nothing written if part of the data is outside 0,...,TOTSIZ-1.
(NUM bytes of data)	
Response:	
STARTP	The starting byte position of the returned data.
NUM	The number of CONF data bytes written.

CODE_Name	
Function: Read CONF instance name	
Data field (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
STRING(INSTNAME)	The name of this CONF object instance. Normally this is "conf".

CODE_Start	
Function: Start CONF data protection	
Data (one byte each unless otherwise specified)	Description
Request:	
PASSW (1-10 bytes)	Arbitrary password data (normally, but not limited to ASCII).
Response:	
(no data)	

CODE_Stop	
Function: Remove CONF data protection	
Data (one byte each unless otherwise specified)	Description
Request:	
PASSW (1-10 bytes)	Arbitrary password data. Removes data protection if identical to PASSW used with CODE_Start.
Response:	
(no data)	If successful - otherwise Err_ParFail

CODE_Info	
Function: Remove CONF data protection	
Data (one byte each unless otherwise specified)	Description
Request:	
(no data)	
Response:	
TOTSIZ	The total size of the CONF data
BLOCKED	1 if data protection is active, 0 otherwise

Note: The CODE_Write messages also has special interpretations. These occur if STARTP=0, NUM=0 and one byte of data is passed (conflicts with normal interpretation). In these cases, the data passed is used as a command code with the following effect:

data = 0: “Set default” - sets entire CONF object to its default values.

data = 1: “Update checksum” - updates CONF internal checksum. This command must be executed after all changes to CONF (CODE_write, CODE_Start or CODE_STOP). If not, the CONF automatically reverts to its default settings at next reset.

When CONF protection is activated (after successful CODE_Start), the CODE_Read, CODE_Write and CODE_Start are all disabled and will result in error code Err_Blocked.

Note that the maximum SVIFT message size, MAX_SM_LEN, also limits NUM to 16..25 depending on header parameter values. It is therefore not recommended to use NUM > 16.

3.1.1.11 Object OTYP_FINFO1

This object represents a highly specialised functionality implemented in fan controllers. Its purpose is to transmit and receive status and temperature information from other fan controllers that it is configured to cooperate with. Each fan controller maintains a table of this information from the cooperating controllers and controls its fan motor based on received information as well as on absence of such. The message format is described here for information only, it is not intended to be used (transmitted from or interpreted by) other types of equipment. Unlike most other objects, it does not respond to CODE_Read or CODE_Name messages.

The object supports the following CODE values:,.

CODE_Info	
Function: Inform on FINFO1 status and temperature	
Data field (one byte each unless otherwise specified)	Description
Request (no response)	
APHYS	The physical address of the unit sending the message.
TEMP	Signed (2’s complement) temperature in centigrades.
FLAGS1	Failure flag bits

3.1.1.12 Object OTYP_FINFO2

This object represents an optional functionality for fan controllers - and enables a more general means of achieving similar functionality as the OTYP_FINFO1 object. Its purpose is to receive requests to run at full speed or to run at a speed corresponding to higher than actual temperature. Each such received message has a life time of 15seconds and its TEMP value is stored in a table that can contain minimum 5 messages during this lifetime. The worst combination of TEMP values from this table, and from the fan controller's own temperature and failure flags, controls the fan speed.

The storage into the described table is optimised to save space by:

- never storing entries that do not have higher TEMP than the controller itself.
- overwriting entries that have same or lower TEMP, but lower remaining lifetime.

Using OTYP_FINFO2, multiple fan units can be setup to "cooperate" in various patterns: to run according to the highest temperature seen by a group of fans or other measured temperature, or to run at full speed when another fan unit is missing or faulty. This cooperation is very flexible, but requires (as opposed to OTYP_FINF1) implementation outside the fan unit itself - should be easy to implement in the supervision equipment.

Unlike most other objects, OTYP_FINFO2 does not respond to CODE_Read or CODE_Name messages.

The object supports the following CODE values:,.

CODE_Write	
Function: Inform on FINFO status and temperature	
Data field (one byte each unless otherwise specified)	Description
Request:	
TEMP	Signed (2's complement) temperature in centigrades. Requests the receiver to run at speed corresponding to this or higher temperature. The value 127 (0x7f) is interpreted as "go maximum speed" - independent from regulation curves.
Response:	
TEMP	Copied from request

The following error responses have special interpretation for this object type:

Err_ReqFail - The storage into the above described table was not successful (no space), the request should be repeated.