CANAL DLL - API specifications:

CanalOpen

long CanalOpen(const char *pConfigStr, unsigned long flags)

Opens a CAN channel.

pConfigStr

Physical device to connect to. This is the place to add device specific parameters and filters/masks. This is a text string. It can be a name, some parameters or whatever the interface creator chooses.The string for usb2can device constists of serial number,can bus speed,mask,filter.Mask and filter can be omited. ***

Example:

ED123456 ; 125 ; 0x12345678 ; 0x98765432
ED123456 ; 1000

for custom CAN bus speed:

ED123456 ; 0 ; tseg2 ; tseg1 ; sjw ; brp ; mask ; filter ***
ED123456 ; 0 ; tseg2 ; tseg1 ; sjw ; brp

flags

device specific flags with a meaning defined by the interface creator.
0x00000001 – enable loopback
0x00000002 – enable silent
0x00000004 - disable auto retransmitions
0x00000008 - enable status messages

returns

handle for open physical interface or < = 0 on error. For an interface where there is only one channel the handle has no special meaning and can only be looked upon as a status return parameter.

*** mask anf filter not implemented yet in currently DLL and usb2can firmware versions

CanalClose

int CanalClose(long handle)

Close the channel and free all allocated resources associated with the channel.

handle

handle for open physical interface.

returns

CANAL error or success code

CanalSend

int CanalSend(long handle, const CANALMSG *pCanMsg)

Send CANAL message. Non blocking function

handle

handle for open physical interface.

pCanMsg

pointer to CANALMSG

returns

CANAL error or success code

CanalBlockingSend

int CanalBlockingSend(long handle, const CANALMSG *pCanMsg, unsigned long timeout)

Send CANAL message. Blocking function

handle
handle for open physical interface.

pCanMsg
pointer to CANALMSG

timeout
timeout in milliseconds. 0 to wait forever.

returns
CANAL error or success code

CanalReceive

int CanalReceive(long handle, CANALMSG *pCanMsg)

Receive CANAL message. Non blocking function

handle
handle for open physical interface.

pCanMsg
pointer to CANALMSG struct.

returns
CANAL error or success code

CanalBlockingReceive

int CanalBlockingReceive(long handle, const CANALMSG *pCanMsg, unsigned long timeout)

Receive CANAL message. Blocking function

handle
handle for open physical interface.

pCanMsg
pointer to CANALMSG

timeout
timeout in milliseconds. 0 to wait forever.

returns
CANAL error or success code

CanalDataAvailable

int CanalDataAvailable(long handle)

Check if there is data available in the input queue for this channel that can be fetched with CanalReceive.

handle
handle for open physical interface.

returns
Number of frames available to read.

CanalGetStatus

int CanalGetStatus(long handle, CANALSTATUS *pCanStatus)

Returns a structure that gives some information about the state of the channel. How the information is interpreted is up to the interface designer. Typical use is for extended error information.

handle
handle for open physical interface.

pCanStatus
Pointer to CANALSTATUS struct.

returns
CANAL error or success code

CanalGetStatistics

int CanalGetStatistics (long handle, CANALSTATISTICS *pCanalStatistics)

Return some statistics about the interface. If not implemented for an interface FALSE should always be returned.

handle
handle for open physical interface.

pCanalStatistics
Pointer to CANALSTATISTICS struct.

returns
CANAL error or success code

CanalSetFilter

int CanalSetFilter (long handle, unsigned long filter)

Set the filter for a channel. There is only one filter available. The CanalOpen call can be used to set multiple filters. If not implemented FALSE should always be returned. Enable filter settings in the open call if possible. If available in the open method this method can be left unimplemented returning false.

handle
handle for open physical interface.

filter
filter for the interface.

returns
CANAL error or success code

CanalSetMask

int CanalSetMask (long handle, unsigned long mask)

Set the mask for a channel. There is only one mask available. The CanalOpen call can be used to set multiple masks. If not implemented FALSE should always be returned. Enable mask settings in the open call if possible. If available in the open method this method can be left unimplemented returning false.

handle
handle for open physical interface.

mask
filter for the interface.

returns
CANAL error or success code

CanalSetBaudrate

int CanalSetBaudrate (long handle, unsigned long baudrate)

Set the bus speed for a channel. The CanalOpen call may be a better place to do this. If not implemented FALSE should always be returned. Enable baudrate settings in the open call if possible. If available in the open method this method can be left unimplemented returning false.

handle
handle for open physical interface.

baudrate
the bus speed for the interface.

returns
CANAL error or success code

CanalGetVersion

unsigned long CanalGetVersion (void)

Get the Canal version. This is the version derived from the document that has been used to implement the interface.

returns
Canal version expressed as an unsigned long.

```
CANAL_MAIN_VERSION  (8)
CANAL_MINOR_VERSION (8)
CANAL_SUB_VERSION   (8)
0
```

CanalGetDllVersion

unsigned long CanalGetDllVersion (void)

Get the version of the interface implementation. This is the version of the code designed to implement Canal for some specific hardware.

returns
DLL version expressed as an unsigned long.

```
DLL_MAIN_VERSION   (8)
DLL_MINOR_VERSION  (8)
DLL_SUB_VERSION    (8)
0
```

CanalGetVendorString

const char * CanalGetVendorString (void)

Get a pointer to a null terminated vendor string for the maker of the interface implementation. This is a string that identifies the constructor of the interface implementation and can hold copyright and other valid information.

returns
Pointer to a vendor string.

example:
usb2can_firmware_version ; usb2can_ hardware_version ; canal_version ; dll_version ; vendor

CanalGetDriverInfo ***

const char * CanalGetDriverInfo(void)

This call returns a documentation object in XML form of the configuration string for the driver. This string can be used to help users to enter the configuration data in an application which allows for this.

returns
Pointer to a configuration string or NULL if no configuration string is available.

*** not imlemented yet

CANAL error codes

CANAL_ERROR_SUCCESS	0	All is OK.
CANAL_ERROR_BAUDRATE	1	Baudrate error.
CANAL_ERROR_BUS_OFF	2	Bus off error
CANAL_ERROR_BUS_PASSIVE	3	Bus Passive error
CANAL_ERROR_BUS_WARNING	4	Bus warning error
CANAL_ERROR_CAN_ID	5	Invalid CAN ID
CANAL_ERROR_CAN_MESSAGE	6	Invalid CAN message
CANAL_ERROR_CHANNEL	7	Invalid channel
CANAL_ERROR_FIFO_EMPTY	8	Noting available to read. FIFO is empty
CANAL_ERROR_FIFO_FULL	9	FIFO is full
CANAL_ERROR_FIFO_SIZE	10	FIFO size error
CANAL_ERROR_FIFO_WAIT	11	
CANAL_ERROR_GENERIC	12	Generic error
CANAL_ERROR_HARDWARE	13	A hardware related fault.
CANAL_ERROR_INIT_FAIL	14	Initialization failed.
CANAL_ERROR_INIT_MISSING	15	
CANAL_ERROR_INIT_READY	16	
CANAL_ERROR_NOT_SUPPORTED	17	Not supported.
CANAL_ERROR_OVERRUN	18	Overrun.
CANAL_ERROR_RCV_EMPTY	19	Receive buffer empty
CANAL_ERROR_REGISTER	20	Register value error
CANAL_ERROR_TRM_FULL	21	
CANAL_ERROR_ERRFRM_STUFF	22	Errorframe: stuff error detected
CANAL_ERROR_ERRFRM_FORM	23	Errorframe: form error detected
CANAL_ERROR_ERRFRM_ACK	24	Errorframe: acknowledge error
CANAL_ERROR_ERRFRM_BIT1	25	Errorframe: bit 1 error
CANAL_ERROR_ERRFRM_BIT0	26	Errorframe: bit 0 error
CANAL_ERROR_ERRFRM_CRC	27	Errorframe: CRC error
CANAL_ERROR_LIBRARY	28	Unable to load library
CANAL_ERROR_PROCADDRESS	29	Unable get library proc address
CANAL_ERROR_ONLY_ONE_INSTANCE	30	Only one instance allowed
CANAL_ERROR_SUB_DRIVER	31	Problem with sub driver call
CANAL_ERROR_TIMEOUT	32	Blocking call timeout
CANAL_ERROR_NOT_OPEN	33	The device is not open.
CANAL_ERROR_PARAMETER	34	A parameter is invalid.
CANAL_ERROR_MEMORY	35	Memory exhausted.

CANAL_ERROR_INTERNAL	36	Some kind of internal program error
CANAL_ERROR_COMMUNICATION	37	Some kind of communication error

CANALMSG

This is the general message structure

unsigned long flags

Flags for the package.

- Bit 0 – if set indicates that an extended identifier (29-bit id) else standard identifier (11-bit) is used.
- Bit 1 – If set indicates a RTR (Remote Transfer) frame.
- Bit 2 – If set indicates that this is an error package. The data byts holds the error information. id is set to zero. For format see CANAL_IDFLAG_STATUS below.
- Bit 3 – Bit 30 Reserved.
- Bit 31 – This bit can be used as a direction indicator for application software. 0 is receive and 1 is transmit.

unsigned long obid

Used by the driver or higher layer protocols.

unsigned long id

The 11-bit or 29 bit message id.

unsigned char count

Number of data bytes 0-8

unsigned char data[8]

Eight bytes of data.

unsigned long timestamp

A time stamp on the message from the driver or the interface expressed in milliseconds. Can be used for relative time measurements.

PCANALSTATISTICS

This is the general statistics structure

unsigned long cntReceiveFrames

Number of received frames since the channel was opened.

unsigned long cntTransmittFrames

Number of frames transmitted since the channel was opened.

unsigned long cntReceiveData

Number of bytes received since the channel was opened.

unsigned long cntTransmittData

Number of bytes transmitted since the channel was opened.

unsigned long cntOverruns

Number of overruns since the channel was opened.

unsigned long cntBusWarnings

Number of bus warnings since the channel was opened.

unsigned long cntBusOff

Number of bus off's since the channel was opened.

CANALSTATUS

unsigned long channel_status

Bit 0 - TX Error Counter.
Bit 1 - TX Error Counter.
Bit 2 - TX Error Counter.
Bit 3 - TX Error Counter.
Bit 4 - TX Error Counter.
Bit 5 - TX Error Counter.
Bit 6 - TX Error Counter.
Bit 7 - TX Error Counter.
Bit 8 - RX Error Counter.
Bit 9 - RX Error Counter.
Bit 10 - RX Error Counter.
Bit 11 - RX Error Counter.
Bit 12 - RX Error Counter.
Bit 13 - RX Error Counter.
Bit 14 - RX Error Counter.
Bit 15 - RX Error Counter.
Bit 16 - Overflow.
Bit 17 - RX Warning.
Bit 18 - TX Warning.
Bit 19 - TX bus passive.
Bit 20 - RX bus passive.
Bit 21 - Reserved.
Bit 22 - Reserved.
Bit 23 - Reserved.
Bit 24 - Reserved.
Bit 25 - Reserved.
Bit 26 - Reserved.
Bit 27 - Reserved.
Bit 28 - Reserved.
Bit 29 - Bus Passive.
Bit 30 - Bus Warning status.
Bit 31 - Bus off status.

Message ID Flags_(CANALMSG)

Each message has some flags set to give information about the events. The flags are defined as follow

Flag	value	Description
CANAL_IDFLAG_STANDARD	0×00000000	Standard message id (11-bit)

CANAL_IDFLAG_EXTENDED	0×00000001	Extended message id (29-bit)
CANAL_IDFLAG_RTR	0×00000002	RTR-Frame
CANAL_IDFLAG_STATUS	0×00000004	This package is an error indication (data holds error code,id=0).
CANAL_IDFLAG_SEND	0×80000000	Reserved for use by application software to indicate send.

CANAL_IDFLAG_SEND may seam strange but can be very useful for software to use as a way to distinguish between sent and received frames.

CANAL_IDFLAG_STATUS can be used by CAN controllers to report status data back to a host or an application. At the moment the following error codes are defined. All status events consist of four data bytes where the first byte tell the status code, the second is the receive error counter, the third the transmit error counter and the fourth byte is reserved and should be set to zero.

Flag	value	Description
CANAL_STATUSMSG_OK	0×00	Normal condition.
CANAL_STATUSMSG_OVERRUN	0×01	Overrun occured when sending data to CAN bus.
CANAL_STATUSMSG_BUSLIGHT	0×02	Error counter has reached 96.
CANAL_STATUSMSG_BUSHEAVY	0×03	Error counter has reached 128.
CANAL_STATUSMSG_BUSOFF	0×04	Device is in BUSOFF. CANAL_STATUSMSG_OK is sent when returning to operational mode.
CANAL_STATUSMSG_STUFF	0×20	Stuff Error.
CANAL_STATUSMSG_FORM	0×21	Form Error.
CANAL_STATUSMSG_ACK	0×23	Ack Error.
CANAL_STATUSMSG_BIT0	0×24	Bit1 Error.
CANAL_STATUSMSG_BIT1	0×25	Bit0 Error.
CANAL_STATUSMSG_CRC	0×26	CRC Error.

ABOUT

CANAL is tightly coupled with the Very Simple Control Protocol, VSCP and the VSCP daemon. This is a protocol constructed for SOHO control situations. The model has been constructed as a two-layer model so that the VSCP Daemon can also be useful for people that are just interested in CAN and not in VSCP. You can find more information about VSCP at http://www.vscp.org.