The CANAL/VSCP Daemon

Version 0.1.5

2005-09-15

The latest version of this document can be downloaded from:

http://www.vscp.org

info@vscp.org

This document is outdated!!!!

Please visit http://www.vscp.org/wiki/doku.php?id=start instead!

A warning...

This document is simply a collection of notes that eventually will become something useful and a document describing the canal daemon. Everything, including the application, is at an early stage of development at the moment so please be patient, report bugs and check for new versions frequently.

Canald what is it?

Canald is the **CAN A**bstraction **L**ayer **D**aemon. Daemon is the name used in the Unix world for a server or background process whilst Service is the most common term in the Windows world.

The Canald server acts as the interface between one or several physical or logical CAN bus devices and one or several clients that make use of these devices. The function of the server is very much like a HUB for Ethernet. It is important to note that even though Canald is designed to interface to CAN devices, devices can be of any type as long as they adopt the canal interface standard.

An example of this is an UDP interface and a logger device that logs triggers or messages or some other noteworthy events to a text file.

The CAN abstraction layer has been constructed to have one programmatic interface which can access different CAN drivers. In this way a program that uses CAN services can be written that will work with several devices and drivers. This was something that was uncommon when this software (canal/canald) was planned and developed.

As of version 0.1.2 the Very Simple Control Protocol daemon has been included in the canal daemon. This means that there also is a VSCP Level II interface into the daemon and that the VSCP UDP Level II broadcast schema is available.

The canald daemon is available both for Linux and Windows.

Development status: Beta

Current information about **canal**, **canald** and **VSCP** (*Very Simple Control Protocol)* can be found at **http://www.vscp.org** and **http://can.sourceforge.net**. There are two mailing lists available on Sourceforge https://sourceforge.net/mail/?group_id=53560 that is about canal (**can_canal**) and VSCP (**can_vscp**) topics.

To subscribe to the **canal list** go to http://lists.sourceforge.net/lists/listinfo/can-canal

To subscribe to the **VSCP list** go to http://lists.sourceforge.net/lists/listinfo/can-vscp

VSCP what is it?

VSCP stands for <u>Very Simple Control Protocol</u> and it is, as the name implies, a very simple protocol indeed. It is simple because it has been developed for use on low end devices such as microcontrollers.

Even though the protocol is very easy to use it is still very capable and can be used in very demanding control situations.

Except for a very well specified message format the protocol supports global unique identifiers for nodes, a register model to give a flexible interface to node configuration and a model for node functionality. This model called EDA which stands for Event, Decision, Action describes how work is carried out by a node. A node can be as complex as needed and do very tough work but to the world it should support an interface following EDA. This is:

1. **Event.** The node should be capable of reacting to VSCP messages. Each message is considered an event. The node itself decides which messages it is interested in except for a very small common subset of base VSCP protocol functionality messages.

2. **Decision**. The node has a hard defined or user configurable **<u>decision matrix</u>**. This matrix tells the node what should be done when a certain event is received.

3. **Action**. The action is that which a node performs and is the outcome of a decision.

Distribution content

The canald distribution comes with a number of files and tools. At the moment most of the tools are mainly for debugging use. In the future it is hoped that there will be more, quality tools included. Feel free to contribute in any way you see fit.

MFC has been used for most of the user interface elements at the moment. This will probably be changed to wxWidgets to ensure that the GUI is fully platform independent.

wxWidgets (http://www.wxwidgets.org/) has been used for most of the user interface elements and also for many console tools to get platform independent code. This means that most code will work on WIN32, UNIX as well as Mac.

canalservice or canald

This is the actual work horse. The daemon or the server. When started it gives instant service. Any canal enabled application can reach services exposed by canald. Note the "it". You have to choose between **canald** or **canalservice**. Canald is an application and canalservice is a standard windows service.

If you want to auto start **either** *canalservice* or *canald*.

canalservice

The service can be installed with

canalservice -i

Uninstalled with

canalservice -u

More info about different switches and the available options can be retrieved by
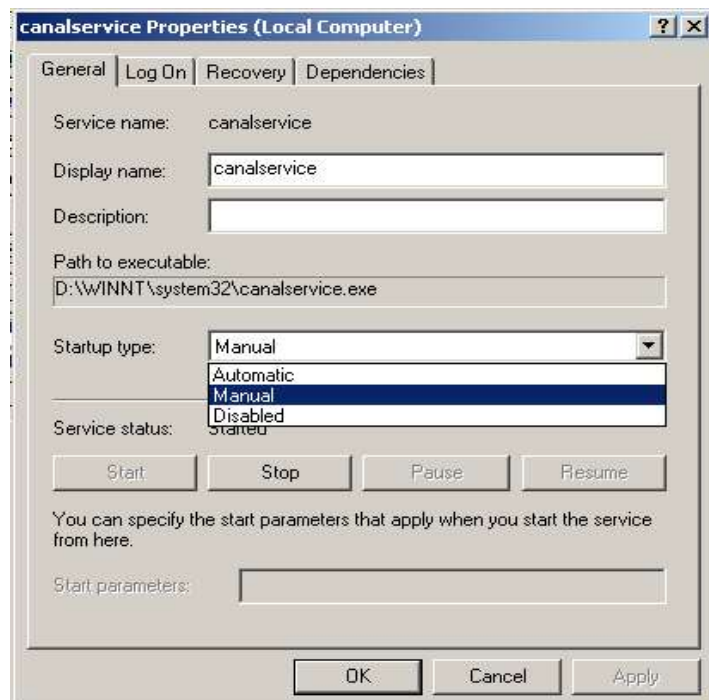
canalservice -?

or

canalservice -h

The service have a variable called start at
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\canalservice that should be set
to 2 (autostart). It is set to 3 (manual) after installation so that you have the option to select which
server/service to use.

This can also be done in the service applet

Where you select automatic if you want the service to start automatically.

The installation program installs the service and marks it to be run automatically. You have to
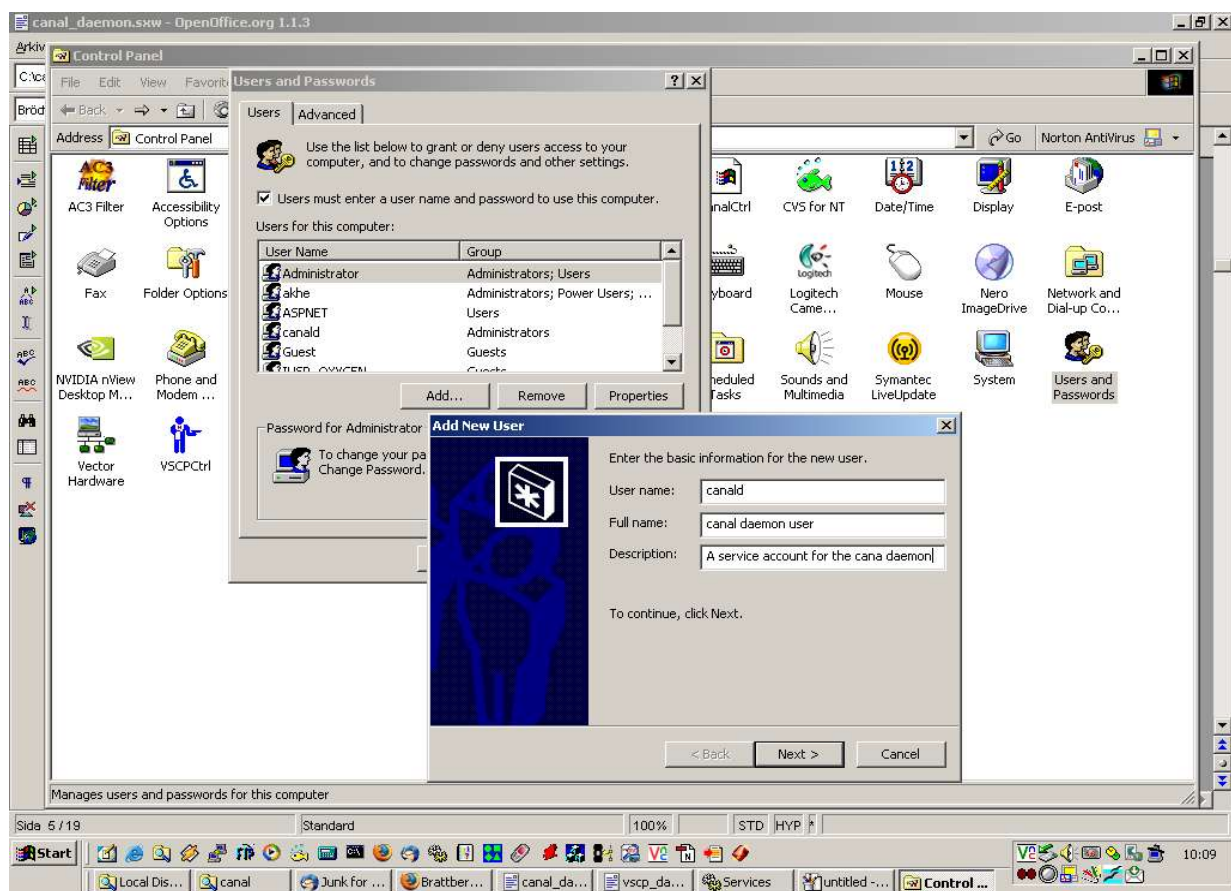
add the user the service runs under manually.

Canalservice needs to be run as a user belonging to the administrator group. Therefore you should add a user "canald" and add this user to the administrators group.

You do this by selecting the Users and Passwords applet in the control panel

and adding the user.

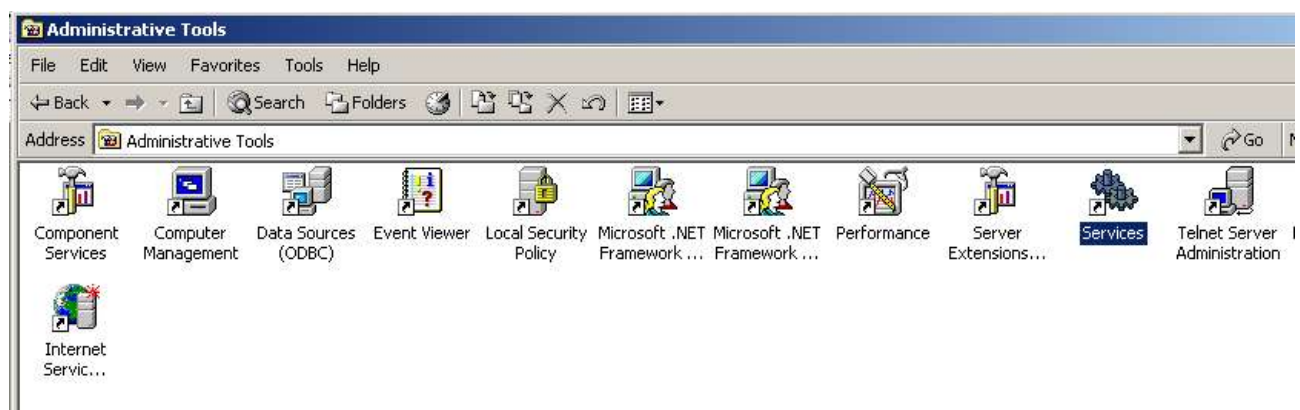After hitting next and entering a password (recommended) this window is shown
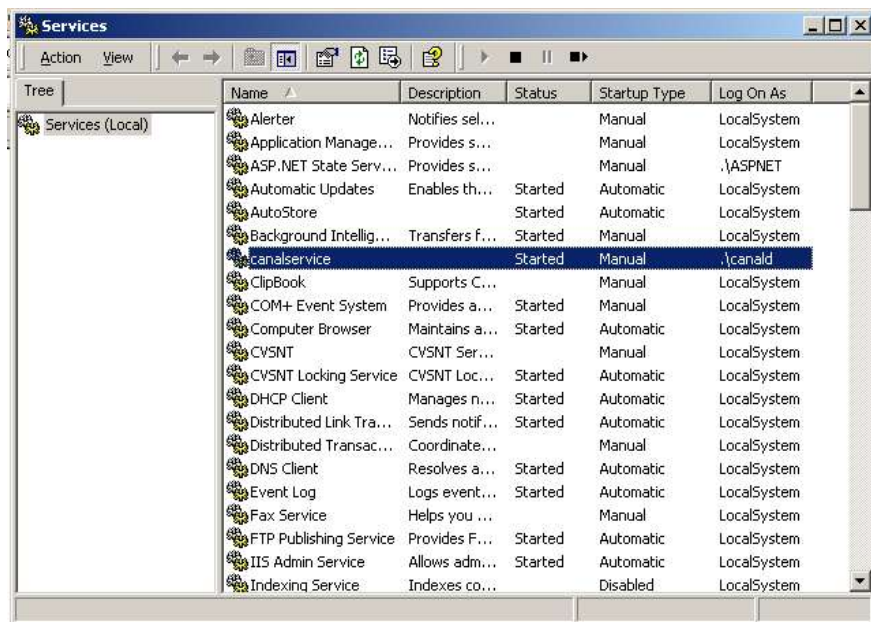
Select others and "Administrators"

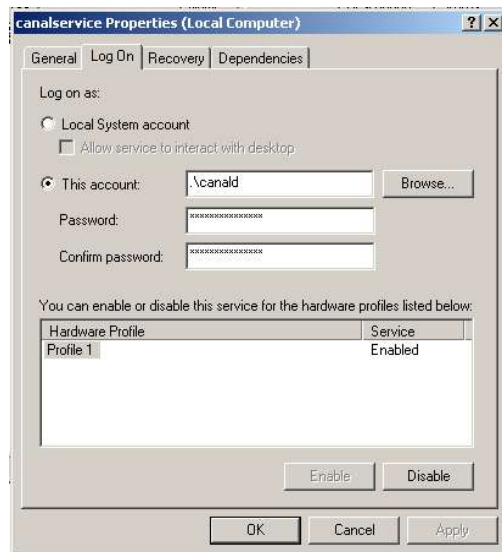After doing that you should open the service applet under the Administrative Tools in the control panel

You select the services applet here

and this window will show up

Right click the canalservice item and select **properties. Select the Logon tab**

Mark "This account" and select the user you added above.

Now you can start the service in the service control applet.

canald

canald is an application version of the canalservice. It can be run on demand or

auto started by creating an entry at
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

Configuration parameters are stored at *HKEY_LOCAL_MACHINE\SOFTWARE\canal\canald*

cw

Cw stands for CanalWorks and it replaces the older CanalDiagnostic. This is work in progress but will eventually and hopefully become a useful CAN/VSCP diagnostic tool on top of Canal. At the moment it allows you to open several channels to the daemon and/or directly to canal interfaces. You can send and receive messages. This will be the centre for future diagnostic tools and the tool is working on both the WIN32 and the Linux platform.

This is how it looks today
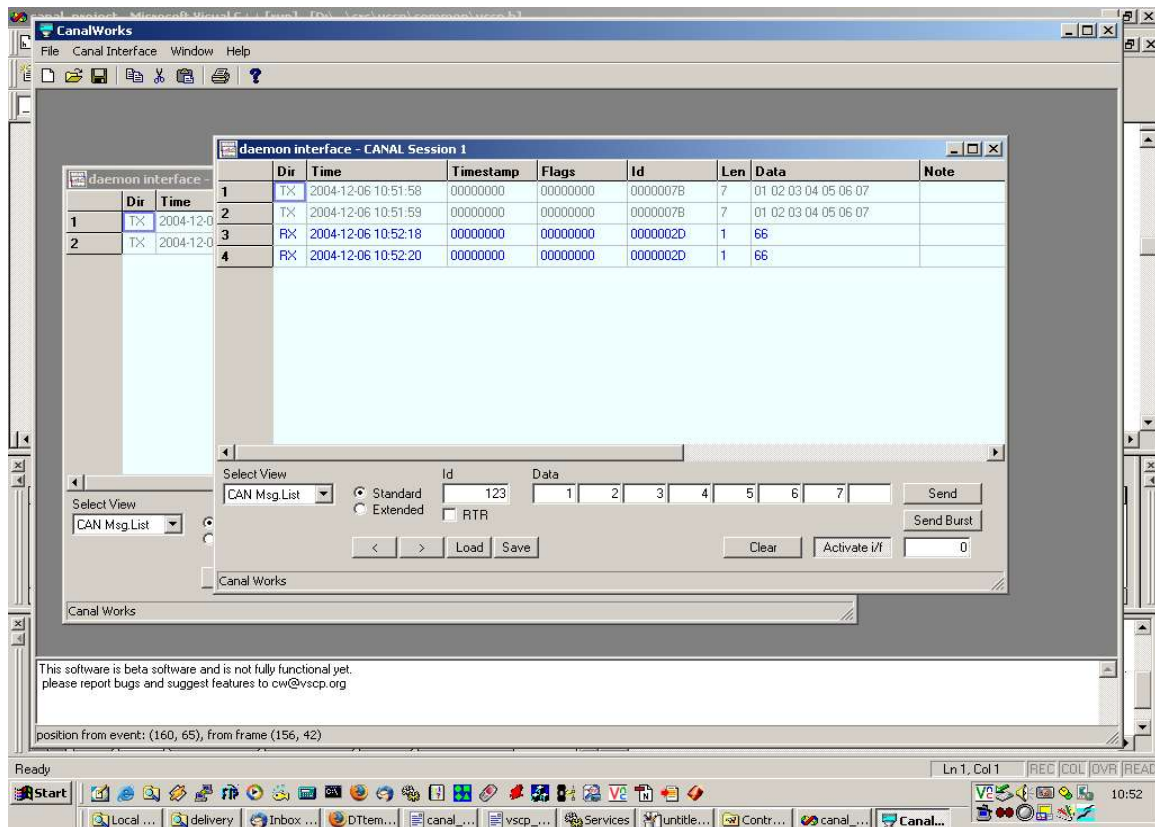
In the above picture there are two sessions open to the canal daemon. You can have as many simultaneous sessions as you need open with a mix of daemon connections and direct device connections.

To open a new connection select File/New List window in the menu or use the toolbar button. This will bring up the interface selection window
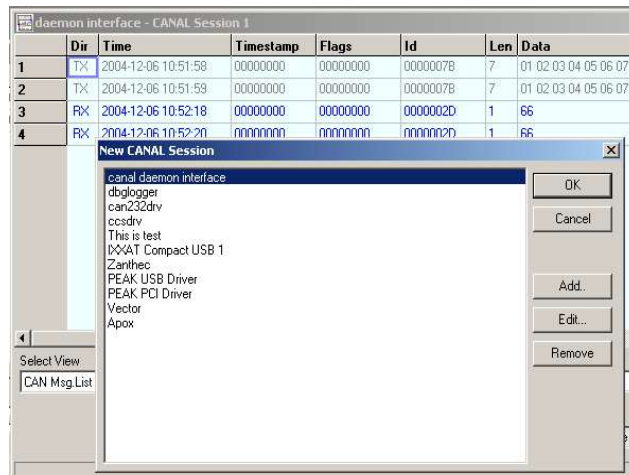
Here you can select the interface you want to open. The first item is always the connection to the daemon and naturally this must be started before it makes sense to connect to this interface.

The rest of the listed interfaces are loaded from the registry. HKEY_LOCAL_MACHINE\SOFTWARE\canal\CanalWorks

You can edit and/or remove all except the daemon interface and you can also add new interfaces.

The **add device** window looks like this
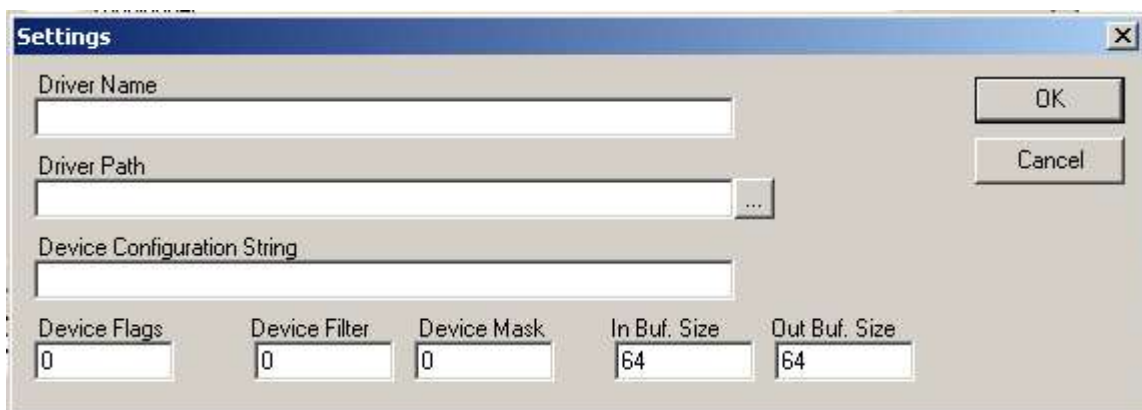
You enter a descriptive **Driver Name** (this is the name that shows up in the list), the path to the driver (the button to the right of the field opens a browse dialog).

The **Device Configuration String** is very important as it gives the parameters on how this particular driver should be opened. The parameters are given as a string with items separated by colon characters. A typical line is:
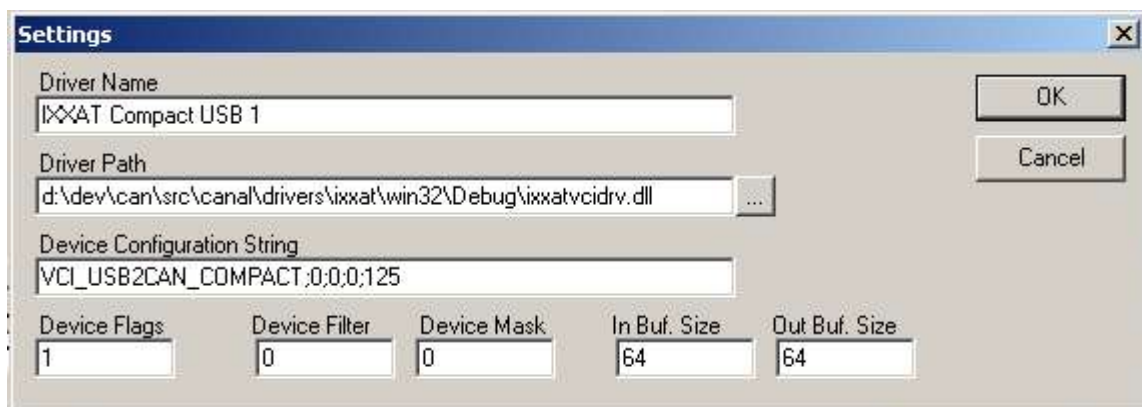
VCI_USB2CAN_COMPACT;0;0;0;125

which is for the IXXATT USB Compact device and sets this device up using the common IXXATT driver in 125 kbps.

The **Device Flags** is a numeric value which is also a device specific field with meaning that varies from driver to driver.

Device Filter and Device Mask settings is the initial filter and mask used on this channel.

The inbuffer and outbuffer sizes are not used at the moment.

With the **edit button** you bring up the same window for the selected item. For IXXATT this could look like
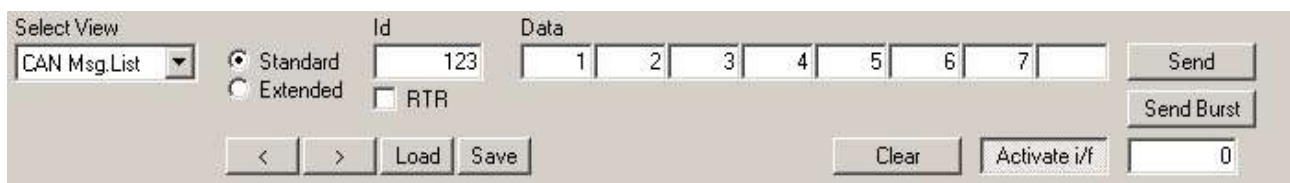

The **remove button** just removes one item.

To select an interface for usage doubleclick it or select it and click OK.

In the message window

You can enter standard/extended messages to send. You may notice that no data length is available. This is because the length is set automatically from available content in the data fields. If you enter a value in the right most field and nothing in the rest a zero will be used for them.

To send a message click the send button. The message will show up in the log area. You can use the

to send multiple messages. Enter a value > 0 in the edit box and that number of messages of the selected type and format will be sent on the interface.

The different views are not yet enabled however VSCP messages, statistics etc. will be added in due course.

Additionally, the **Load** and **Save** including the **forward**, **back** buttons are not enabled but will make it possible to have a set of predefined messages available for easy reference.

canaldll

This is the dll interface to the canal daemon. This is a standard canal driver, the only difference being that it connects its users to the canald instead of a device.

There is also a library for inclusion into MSVC projects included in the distribution.

Check the canal specification document for information on the exported functions that are available and the full documentation.

If you work in C++ we recommend that you use the CanalSharedMemLevel1 class which is defined in canalshmem_level1_*win32.h/canalshmem_level1_win32.cpp* for the WIN32 platform and in *canalshmem_level1_unix.h/canalshmem_level1_unix.cpp* platform.

For use with Visual Basic, VBS, ASP or similar, the CANAL Active X control is probably the easiest solution.

Open a CANAL device and get a handle to it.

@param pDevice Driver parameter string

@param flags - Give extra info to the CANAL i/F.

@return Handle of device or -1 if error.

*long WINAPI EXPORT CanalOpen(char *pDevice, unsigned long flags);*

Close a CANAL channel.

@param handle - Handle to open physical CANAL channel.

@return zero on success or error-code on failure.

int WINAPI EXPORT CanalClose(long handle);

Get CANAL DLL supported level

@return level for CANAL dll implement ion.

unsigned long WINAPI EXPORT CanalGetLevel(long handle);

Send a message on a CANAL channel.

The instruction should block unless instructed not todo so when the interface was opened.

@param handle - Handle to open physical CANAL channel.

@param pCanMsg - Message to send.

@return zero on success or error-code on failure.

int WINAPI EXPORT CanalSend(long handle, PCANALMSG pCanalMsg);

Receive a message on a CANAL channel.

The instruction should block unless instructed not to

do so when the interface was opened.

@param handle - Handle to open physical CANAL channel.

@param pCanMsg - Message to send.

@return zero on success or error-code on failure.

int WINAPI EXPORT CanalReceive(long handle, PCANALMSG pCanalMsg);

Check a CANAL channel for message availability.

@param handle - Handle to open physical CANAL channel.

@return zero if no message is available or the number of messages waiting to be received.

int WINAPI EXPORT CanalDataAvailable(long handle);

Get status for a CANAL channel

@param handle Handle to open physical CANAL channel.

@param pCanStatus Pointer to a CANAL status structure.

@return zero on success or error-code on failure.

int WINAPI EXPORT CanalGetStatus(long handle, PCANALSTATUS pCanalStatus);

Get statistics for a CANAL channel

@param handle Handle to open physical CANAL channel.

@param pCanStatistics Pointer to a CANAL statistics structure.

@return zero on success or error-code on failure.

int WINAPI EXPORT CanalGetStatistics(long handle, PCANALSTATISTICS pCanalStatistics);

Set the mask for a CANAL channel

@param handle Handle to open physical CANAL channel.

@param pCanStatistics Pointer to a CANAL statistics structure.

@return zero on success or error-code on failure.

int WINAPI EXPORT CanalSetFilter(long handle, unsigned long filter);

Set the filter for a CANAL channel

@param handle Handle to open physical CANAL channel.

@param mask for

@return zero on success or error-code on failure.

int WINAPI EXPORT CanalSetMask(long handle, unsigned long mask);

Set the baud rate for a CANAL channel

@param handle Handle to open physical CANAL channel.

@param baud rate Baud rate for the channel

@return zero on success or error-code on failure.

int WINAPI EXPORT CanalSetBaudrate(long handle, unsigned long baud rate);

Get CANAL version

@return version for CANAL i/f.

unsigned long WINAPI EXPORT CanalGetVersion(void);

Get CANAL DLL version

@return version for CANAL dll implement ion.

unsigned long WINAPI EXPORT CanalGetDllVersion(void);

Get CANAL vendor string

@return version for CANAL dll implemention.

*const char * WINAPI EXPORT CanalGetVendorString(void);*

canalocx

This is an OCX control that provides an automation interface to CANAL. It can connect to a device driver directly or to the daemon. This control can be utilised for Visual Basic, ASP and VBA interfacing to canal.

There is a Visual Basic application included in the distribution that demonstrates its usage.
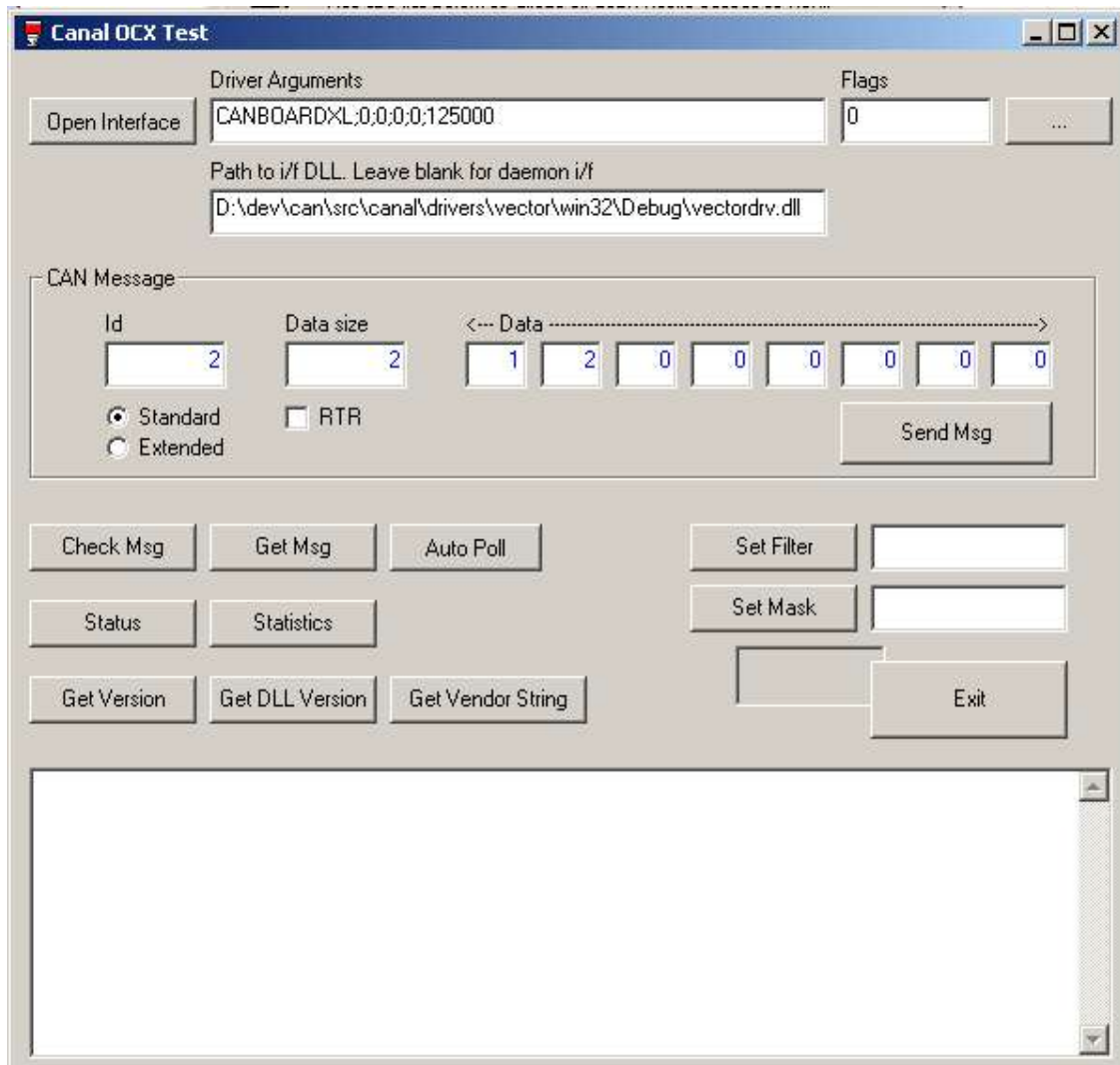
Properties

Cstring DeviceDllPath

Set the path to the driver DLL or leave empty to use the daemon interface.

short DeviceInBufferSize

Not used at the moment.

short DeviceOutBufferSize

Not used at the moment.

long DllVersion (Read Only)

Returns the version of the interface DLL.

long filter

Set the filter for the interface.

log mask

Set the mask for the interface.

short MsgData(index)

This property is part of the message structure.

MsgData holds the eight data positions of the message (index=0 – 7)

long MsgFlags

This property is part of the CAN message structure.

This property holds the flags for the message. These flags indicate whether the message uses an extended or a standard id, is a remote frame etc. The bMsgStarnardId, bMsgExtendedId and bMsgRTR are easier ways to check/set the flags.

long MsgId

This property is part of the CAN message structure.

This is the id for the message.

long MsgObId

This property is part of the CAN message structure.

This is information supplied by the daemon which indicates the interface that originated the message. It has no meaning when sending a message.

short MsgSizeData

This property is part of the CAN message structure.

This is the data size of the message and can be a value between 0 and 8.

long MsgTimeStamp

This property is part of the CAN message structure.

If the driver timestamps messages this time stamp can be read from this property. It has no meaning when sending a message.

long StatCntBusOff (Read Only)

This property is part of the CAN statistics structure.

This is a counter for bus off states.

long StatCntBusWarnings (Read Only)

This property is part of the CAN statistics structure.

This is a counter for bus warning states.

long StatCntOverruns (Read Only)

This property is part of the CAN statistics structure.

This is a counter for overruns.

long StatCntReceiveData (Read Only)

This property is part of the CAN statistics structure.

This is a counter for interface received data.

long StatCntReceiveFrames (Read Only)

This property is part of the CAN statistics structure.

This is a counter for interfaces received frames.

long StatCntTransmitData (Read Only)

This property is part of the CAN statistics structure.

This is a counter for interface transmitted data.

long StatCntTransmitFrames (Read Only)

This property is part of the CAN statistics structure.

This is a counter for interface transmitted frames.

long StatusState (Read Only)

This is the interface state. The interpretaion is device specific.

BSTR VendorString (Read Only)

This property holds the interface vendor string.

long Version (Read Only)

This property holds the CANAL version.

bMsgExtendedId

If you want to send a message with an extended id set this property to true. If you receive a message you can check this property to see if it has an extended id.

You can of course use the MsgFlags directly if you want.

bMsgRTR

If you want to send a remote frame message set this property to true. If you receive a remote frame message this property will be set to true..

You can of course use the MsgFlags directly if you want.

bMsgStarnardId

If you want to send a message with a standard id set this property to true. If you receive a message you can check this property to see if it has a standard id.

You can of course use the MsgFlags directly if you want.

Methods

BOOL Close

Close the interface. Returns true on success.

Short DataAvailable

Check if there is data available. Returns the number of messages if data is available otherwise 0 is returned.

BOOL NOOP

A dummy message. Can be used to test if the daemon is present.

BOOL Open(LPCTSTR szDevice, long flags)

Open the interface. SzDevice contains theinterface parameters or the device string. Flags are switches for the device. Both are device specific.

The **DeviceDllPath** property should be set before using this method or be left blank to use the daemon interface.

Ex1

Canal1.DeviceDllPath = "D:\winnt\system32\apoxdrv.dll"

If (Canal1.Open("FTMZMX5K;125", 0)) Then

....

End If

Opens the Apox Controls USB Driver with a bit rate of 125 kbps.

Ex2

Canal1.DeviceDllPath = ""

If (Canal1.Open(txtArgs.Text, Val(txtFlags.Text))) Then

....

End If

Open an interface to the daemon.

BOOL Receive();

Receive one message for the interface if one is available. True is returned on success false on failure.

The message is written into **MsgFlags**, **MsgId**, **MsgData(index)**, **MsgSizeData**, **MsgTimeStamp**, **MsgObId** properties.

Ex1

```
If (Canal1.Receive) Then
        ' Message id
         logstr = "message received: id=" + Str(Canal1.MsgId)


        ' Flags
        logstr = logstr + " flags=" + Str(Canal1.MsgFlags)


        ' Extended/Standard Message flag
        If (Canal1.bMsgExtendedId) Then
                logstr = logstr + " EXT-ID "
        Else
                logstr = logstr + " STD-ID "
        End If


        ' RTR
        If (Canal1.bMsgRTR) Then
            logstr = logstr + " RTR "
        End If


        ' Datasize
        logstr = logstr + " dataSize=" + Str(Canal1.MsgSizeData)


        If (Canal1.MsgSizeData > 0) Then
           logstr = logstr + " data=["
        End If


        ' Data
        For i = 0 To Canal1.MsgSizeData - 1
                logstr = logstr + " " + Str(Canal1.MsgData(i))
        Next
```

```
        If (Canal1.MsgSizeData > 0) Then
               logstr = logstr + " ]"
        End If


        ' Timestamp
        logstr = logstr + " timestamp=" + Str(Canal1.MsgTimeStamp)


        ' obid
        logstr = logstr + " obid=" + Str(Canal1.MsgObid)


         LogMessage (logstr)
    Else
        LogMessage ("Failed to receive message (possibly empty
receive queue).")
    End If
```

BOOL Send();

Send a message on the interface. True is returned on success false on failure.

Before sending the message fill in the **MsgFlags**, **MsgId**, **MsgData(index)**, **MsgSizeData**, **MsgTimeStamp**, **MsgObId** properties.

Ex1
```
        ' set message data
        Canal1.MsgId = Val(txtMsgId.Text)
        txtMsgId.Text = Canal1.MsgId


        If (OptionExtendedFrame.Value = True) Then
               Canal1.bMsgExtendedId = True
        Else
               Canal1.bMsgStandardId = True
        End If


        If (CheckRtr.Value = 1) Then
               Canal1.bMsgRTR = True
        End If


        ' size
        If ((Val(txtSizeData.Text) < 8) And (Val(txtSizeData.Text)
> -1)) Then
```

```vb
        Canal1.MsgSizeData = Val(txtSizeData.Text)
Else
        Canal1.MsgSizeData = 0
End If
txtSizeData.Text = Canal1.MsgSizeData ' feedback

For i = 0 To Canal1.MsgSizeData - 1
        Canal1.MsgData(i) = Val(txtMsgData(i).Text)
        txtMsgData(i).Text = Canal1.MsgData(i) ' feedback
Next

If (Canal1.Send) Then
        LogMessage ("Message sent.")
Else
        LogMessage ("Failed to send message.")
End If
```
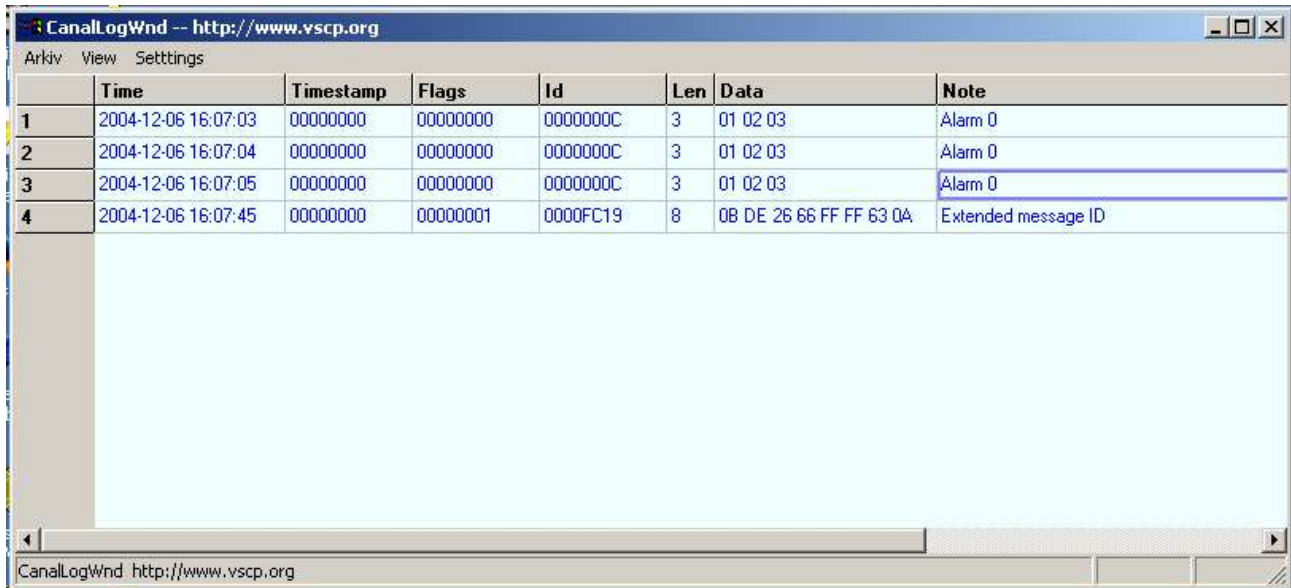
Events

There are no events at the moment.

loggerWnd

This is a client to canald that logs incoming messages in a window. The CW application has all the features that this application has and more. The CW application is probably the first choice however this application can be valuable as an example on how to use the interface class.

CanCmd

This is a simple console based application that can be very useful. With it you can send and receive both CAN and VSCP messages (Level I and Level II). The following is a brief description of cancmd

First of all you can get help with **/h**

If you type

cancmd /h

you get the help screen.

```
Usage: cancmd [/i <num>] [/d <str>] [/p <num>] [/c <num>] [/t
<num>] [/n <nu
[/l <num>] [/g <str>] [/m <str>] [/h] [/v] [/e] [/r] [/z]
  /i, --id=<num>          CAN id (required for CAN package).
  /d, --data=<str>        Comma separated CAN or VSCP data enclosed
in quotes
                          (Required for CAN and VSCP packages).
  /p, --priority=<num>    CAN or VSCP priority (defaults to 0).
  /c, --class=<num>       VSCP Level I and Level II class
  /t, --type=<num>        VSCP Level I and Level II type
  /n, --count=<num>       # of messages to receive send
  /l, --level=<num>       Selects protocol level (1 or 2)
  /g, --guid=<str>        Comma separated VSCP GUID address.
  /m, --value=<str>       Enter VSCP value as data.
  /h, --help              Shows this message
  /v, --verbose           Vebose mode
  /e, --extended          Marks a package with an extended id.
  /r, --receive           Receive mode.
  /z, --test              Interface test mode (for canald developers
only).
```
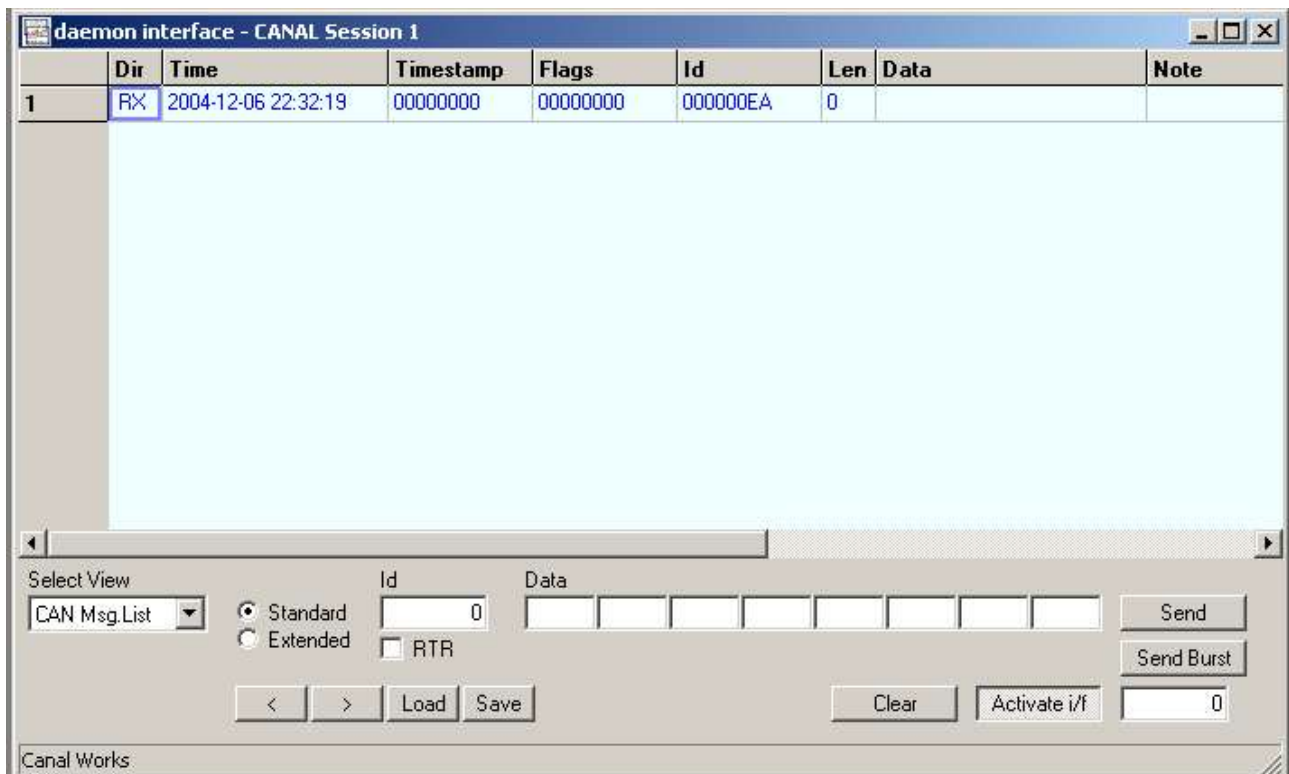
Note that this is under windows. On Unix "-" is used instead on "/" while the long form of the switch/option is the same on both platforms. In the rest of this text we use the long form.

The default for cancmd is to send a message so

cancmd –id=234

Will send a CAN message with no data and id = 234 through the daemon interface.

A message with a standard id was sent. If we want an extended id we use:

cancmd –id=234 --extended

If we also want to send data we can use:

cancmd –id=234 –extended –data=”1,2,3,4,5,6,7,8”

If we want to burst this data out ten times just add –**count:**

cancmd –id=234 –extended –data=”1,2,3,4,5,6,7,8” --count=10

You can add **–verbose** if you want more informative text.

--test is a simple developers test for the daemon software that probably does not have any practical usage other then for developers.
.
To receive data use:
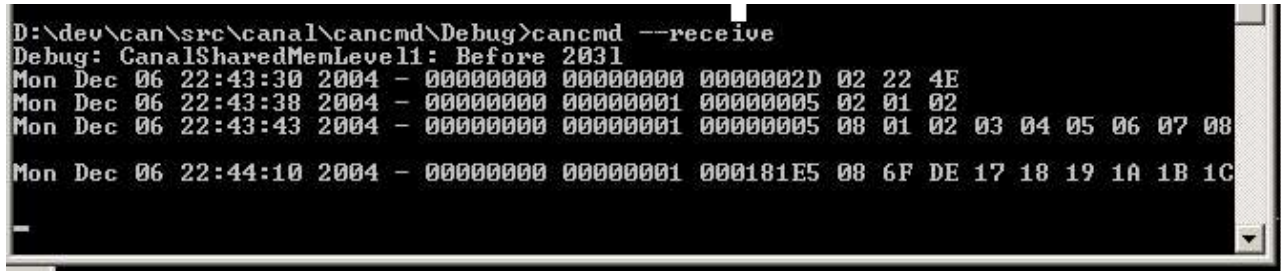
cancmd –receive

Data will now be received until aborted with ctrl+c

The output is something like this:

You can also here add **–count=n** to just receive n messages.

The rest of the command switches are for VSCP usage and are described in the VSCP documentation.

m switch

The m switch let you enter VSCP data values is a simple way. It has the following format

-m"x,t,v"

where

        x - Data Encoding. (see VSCP specification)

        t - Format (0 = default, typically Kelvin, Celsius etc.).

        v - value (ex. 2.45)

testifdll

This is a test application for the canaldll daemon interface dll. It can be used as a sample of how to interface to the daemon using the dll. Note that this example uses dynamic loading of the dll. It is often easier to link your application with the link library.

testif

A test application for the internal message interface to canald. You should probably use **CW** instead of this application. The application can still be useful as an example on how to use the interface class.

Canallogger

CANAL Driver: canallogger.dll(win32), canallogger.so(Linux)

Include library for msvc: canallogger.lib
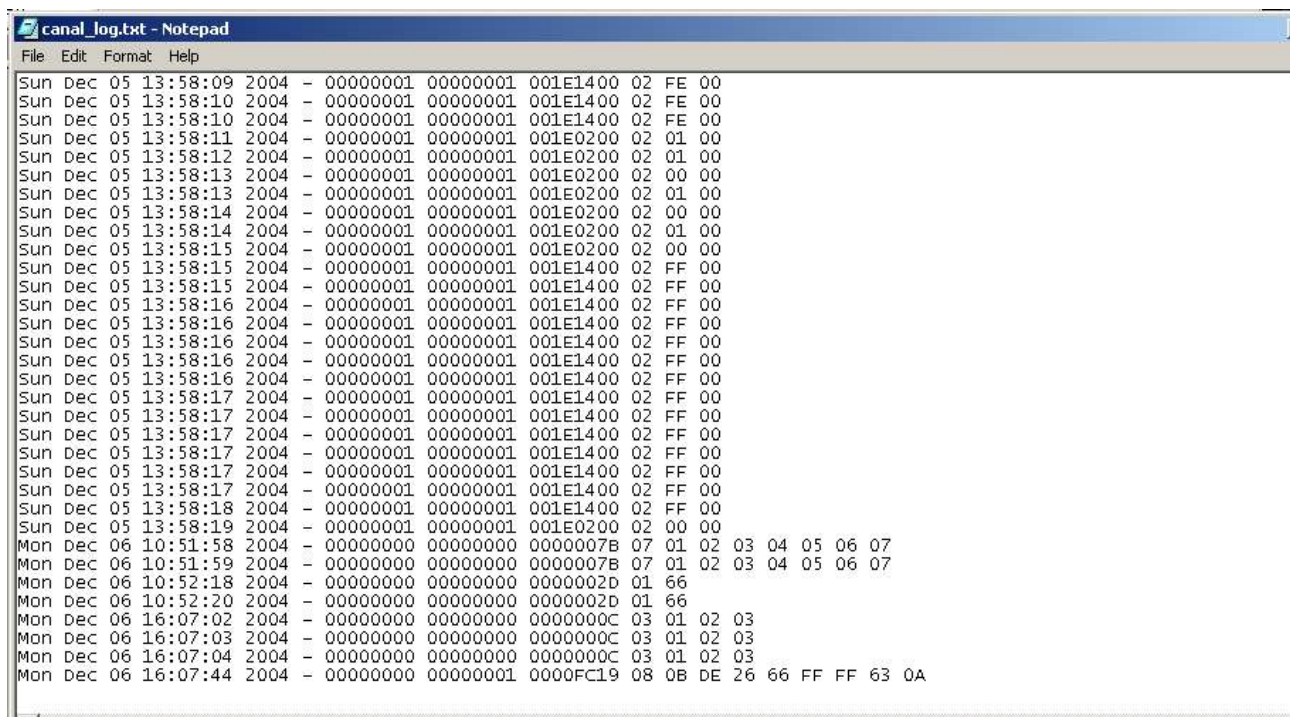
Device driver for diagnostic logging. It allows you to log CAN traffic to a text file. Several drivers can be loaded with different output files and using different filter/masks.

DriverString: The absolute path including the file name to the file that log data will be written to.

Flags: **0** – Append data to an existing file (Create if not available).

1 – Create a new file or rewrite data on an old file.

Status return: The CanalGetStatus call returns the status structure with the *channel_status* member having the following meaning:

0 is always returned.

Platforms: **WIN32**, **Linux**

can232drv

This driver interface is for the can232 adapter from Lawicel *(http://www.lawicel.com* or *http://www.can232.com*). This is a low cost CAN adapter that connects to one of the serial communication ports on a computer. The driver can handle the adapter in both polled and non polled mode, whichhandled transparently to the user. It is recommended however that the following settings are made before real life use.

1. Set the baud rate for the device to 115200. You do this with the **U1** command. This is the default baud rate used by this driver.
2. Set auto poll mode by issuing the **X1** command.
3. Enable the time stamp by issuing the **Z1** command.

DriverString: The driver string has the following format (note that all values can be entered in either decimal or hexadecimal form (for hex precede with 0x).

comport;baudrate;mask;filter;bus-speed;btr0;btr1

comport is the serial communication port to use.(for example 1,2,3....).

baudrate is a valid baudrate for the serial interface (for example. 9600).

mask is the mask for the adapter. Read the Lawicel CAN232 manual on how to set this. It is not the same as for CANAL.

filter is the filter for the adapter. Read the Lawicel CAN232 manual on how to set this. It is not the same as for CANAL.

bus-speed is the speed or the CAN interface. Valid values are

10	10Kbps
20	20Kbps
50	50Kbps
100	100Kbps
125	125Kbps
250	250Kbps

500	500Kbps
800	800Kbps
1000	1Mbps

btr0/btr1 Optional. Instead of setting a bus-speed you can set the SJA1000 BTR0/BTR1 values directly. If both are set the bus_speed parameter is ignored. This link can be a help for data http://www.port.de/engl/canprod/sv_req_form.html

If no device string is given COM1 / ttyS0 will be used. Baud rate will be set to 115200 baud and the filter/mask to fully open. The CAN bit rate will be 125Kbps.

Flags: Not used, set to 0.

Status return: The CanalGetStatus call returns the status structure with the *channel_status* member having the following meaning:

Bit 0-22	Reserved
Bit 23	Transmit buffer full
Bit 24	Receive Buffer Full
Bit 25-27	Reserved.
Bit 28	Bus Active.
Bit 29	Bus passive status.
Bit 30	Bus Warning status.
Bit 31	Bus off status.

Platforms: **WIN32**, **Linux**

Example

5;115200;0;0;1000

Uses COM5 at 115200 with filters/masks open and with 1Mbps CAN bit rate.

1;57600;0;0;0;0x09;0x1C

Uses COM1 at 57600 with filters/masks open and with bitrate set to 50Kbps using btr0/btr1

There is an application, can232drvTest, included with the driver that can be used for testing.

Commdrv

This is a device driver that can be used to set up a serial link and exchange CAN packet data through this link.

DriverString:	The driver string has the following format (note that all values can be entered in either decimal or hexadecimal form (for hex precede with 0x).
	comport;baudrate
	comport is the serial communication port to use.(for example 1,2,3....).
	baudrate is a valid baudrate for the serial interface (for example. 9600).
Flags:	Not used set to 0.
Status return:	The CanalGetStatus call returns the status structure with the *channel_status* member having the following meaning:
	Currently undefined
Platforms:	**WIN32**, **Linux**

This driver is not yet included in the distribution and is only available in the source distribution.

genericdll

This is a generic driver for canald that can be used as a template for new driver development.

tcpdrv

This is a device driver that can be used to set up a TCP link and exchange CAN packet data through this link.

DriverString:	The driver string has the following format (note that all values can be entered in either decimal or hexadecimal form (for hex precede with 0x).
	remote_url;remote_port;timeout;local_ip;local_port
	remote_url remote url to connect to. Either as an ip address or standard url.
	remote_port the port to connect to at the remote URL.
	timeout If no packages are received during this time the link will be closed. Set to zero for no timeout.
	local_ip If local_ip is given then remote connections will be accepted on this interface.
	local_port This is the port used for remote connections.
Flags:	Not used set to 0.
Status return:	The CanalGetStatus call returns the status structure with the *channel_status* member having the following meaning:
	Currently undefined
Platforms:	**WIN32**, **Linux**

This driver is not yet included in the distribution and is only available in the source distribution.

udpdrv

This is a device driver that can be used to set up a UDP link and exchange CAN packet data through this link.

DriverString: The driver string has the following format (note that all values can be entered in either decimal or hexadecimal form (for hex precede with 0x).

remote_url;remote_port;timeout;local_ip;local_port

remote_url remote url to connect to. Either as an ip address or standard url.

remote_port the port to connect to at the remote URL.

t**imeout** If no packages are received during this time the link will be closed. Set to zero for no timeout.

local_ip If local_ip is given then remote connections will be accepted on this interface.

local_port This is the port used for remote connections.

Flags: Not used set to 0.

Status return: The CanalGetStatus call returns the status structure with the *channel_status* member having the following meaning:

Currently undefined

Platforms: **WIN32**, **Linux**

This driver is not yet included in the distribution and is only available in the source distribution.

canpiedrv

This is work in progress aiming for a **CANPie** (http://sourceforge.net/projects/canpie/) canald driver.

ccsdriver

Device driver for for the CCS (**CC Systems AB**, http://www.cc-systems.se) PCI CAN card.

There are proprietary libraries and include files needed to build this library from source. The rest of the source is available.

DriverString: Not used.

Flags: Not used set to 0.

Status return: The CanalGetStatus call returns the status structure with the *channel_status* member having the following meaning:

Currently undefined

Platforms: **WIN32**, **Linux**

apoxdrv

CANAL Driver: apoxdrv.dll(win32), apoxdrv.so(Linux)

Include library for msvc: apoxdrv.lib

This is a device driver for the Apox Controls (http://www.apoxcontrols.com/usbcan.htm) USB adapter.

DriverString: The driver string has the following format (note that all values can be entered in either decimal or hexadecimal form (for hex preceded with 0x).

serial;bus-speed

<u>**serial**</u> This is the serial number for the adapter in use.

<u>**bus_speed**</u> This is the speed for the CAN bus. It can be given as

125	for 125Kbps
250	for 250Kbps
500	for 500Kbps
1000	for 1Mbps

Flags: Not used set to 0.

Status return: The CanalGetStatus call returns the status structure with the *channel_status* member having the following meaning:

Bit 0-7   TX error counter

Bit 8-15   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 21-28 Reserved.

Bit 29   Bus Passive.

Bit 30   Bus Warning status

Bit 31   Bus off status

Platforms: **WIN32**, **Linux**

ixxatvcidrv

This is a device driver for the **IXXAT VCI** Driver interface
(http://www.ixxat.de/english/index.html).

DriverString: The driver string has the following format (note that all values can be entered in either decimal or hexadecimal form (for hex precede with 0x).

board;channel;filter;mask;bus-speed;btr0;btr1

board should be one of:
VCI_IPCI165 or 0 for **iPC-I 165, ISA slot**
VCI_IPCI320 or 1 for **iPC-I 320, ISA slot**
VCI_CANDY or 2 for **CANdy320, LPT port**
VCI_PCMCIA or 3 for **tinCAN, pc card**
VCI_IPCI386 or 5 for **iPC-I 386, ISA slot**
VCI_IPCI165_PCI or 6 for **iPC-I 165, PCI slot**
VCI_IPCI320_PCI or 7 for **iPC-I 320, PCI slot**
VCI_CP350_PCI or 8 for **iPC-I 165, PCI slot**
VCI_PMC250_PCI or 9 for special hardware from PEP
VCI_USB2CAN or 10 for **USB2CAN, USB port**
VCI_CANDYLITE or 11 for **CANdy lite, LPT port**
VCI_CANANET or 12 for **CAN@net, ethernet**
VCI_BFCARD or 13 for **byteflight Card, pc card**
VCI_PCI04_PCI or 14 for **PC-I 04, PCI slot, passive**
VCI_USB2CAN_COMPACT or 15 for **USB2CAN compact**, USB port
VCI_PASSIV or 50 for **PC-I 03, ISA slot, passive**

channel Is a value from 0 an up indicating the CAN channel on the selected board.

filter Is the hardware dependent filter for this board hardware.
Note that this filter may work in a different way then the CANAL filter.

mask Is the hardware dependent mask for this board hardware.

Note that this filter may work in a different way then the CANAL filter.

bus-speed One of the predefined bit rates can be set here

10 for 10 Kbps

20 for 20 Kbps

50 for 50 Kbps

100 for 100 Kbps

125 for 125 Kbps

250 for 250 Kbps

500 for 500 Kbps

800 for 800 Kbps

1000 for 1000 Mobs

btr0/btr1 Value for bit rate register 0/1. If btr value pairs are given then the bus-speed parameter is ignored.

Flags:

bit 0

0	11 bit identifier mode
1	29 bit identifier mode

bit 1

0	High speed
1	A low speed-bus connector is used
	(if provided by the hardware)

bit 2

0	Filter our own TX messages from our receive..
1	All sent CAN objects appear as received CAN objects in the rx queues.

bit 3

0	Active Mode.
1	Passive mode: CAN controller works as passive CAN node (only monitoring) therefore it does not send any acknowledge bit for CAN objects sent by another CAN node.

bit 4

0	No error report objects.
1	Error frames are detected and reported as CAN objects via the rx queues

Status return: The CanalGetStatus call returns the status structure with the *channel_status* member having the following meaning:

Bit 0-1	Reserved.
Bit 2	RemoteQueOverrun
Bit 3	CAN-TX pending
Bit 4	CAN-Init-Mode.
Bit 5	CAN-Data-Overrun.
Bit 6	CAN-Error-Warning-Level.
Bit 7	CAN_Bus_Off_status.
Bit 8-15	Reserved.
Bit 16	29-bit id.
Bit 17	Low speed-bus connector.
Bit 18	All sent CAN objects appear as received CAN objects.
Bit 19	Passive mode
Bit 20	Error frames are detected and reported.
Bit 21-29	Reserved.
Bit 30	Bus Warning status (repeated from bit 6)
Bit 31	Bus off status (repeated from bit 7)

Platforms: **WIN32**

The driver uses proprietary include files and libraries from IXXAT which are not included in the distribution.

Examples

VCI_USB2CAN_COMPACT;0;0;0;125

uses the **USB2CAN compact** USB adapter with a bus-speed on 125kbps and the first and only channel of this adapter with filter/mask fully open. This can also be set as [MVM what does first and only mean]

15;0;0;0;125

PEAK

This is a device driver for the PEAK family of cards
(http://www.port.de/engl/canprod/hardware.html).

DriverString: The driver string has the following format (note that all values can be

entered in either decimal or hexadecimal form (for hex precede with 0x).

For non PnP boards/adapters

board;bus-speed;hwtype;port;irq;channel;filter;mask

else

board;bus-speed;channel;filter;mask

<u>**board**</u> This is one of the following

<u>CANDONGLE or 0</u> for **LPT port adapter**

<u>CANDONGLEPRO or 1</u> for **LPT port adapter (PRO version)**

<u>CANISA or 2</u> for **ISA adapter**

<u>CANPCI or 3</u> for **1 channel PCI adapter**

<u>CANPCI2 or 4</u> for **2 channel PCI adapter**

<u>CANUSB or 5</u> for **USB adapter**

<u>**bus_speed**</u> This is the speed for the CAN bus. It can be given as

5 for 5 Kbps

10 for 10 Kbps

20 for 20 Kbps

50 for 50 Kbps

100 for 100 Kbps

125 for 125 Kbps

250 for 250 Kbps

500 for 500 Kbps

800 for 800 Kbps

1000 for 1000 Mobs

<u>hwtype</u>

Only valid for non PNP cards

CANDONGLE

    2 - dongle

    3 - epp

    5 - sja

    6 - sja-epp

CANDONGLEPRO

    7 - dongle pro

    8 - epp

CANISA

    1 - ISA

    9 – SJA

port

For ISA and parallel port adapters this is the hardware
port address

irq

This is the interrupt to use for non PNP devices.

channel

Is a value from 0 an up indicating the CAN channel on the
selected board.

filter

Is the hardware dependent filter for this board hardware.
Note that this filter may work in a different way than
the CANAL filter.

mask

Is the hardware dependent mask for this board hardware.
Note that this filter may work in a different way than
the CANAL filter.

Flags:	**bit 0**		
		0	**11 bit identifier mode**
		1	**11/29 bit identifier mode**

Status return: The CanalGetStatus call returns the status structure with the *channel_status* member having the following meaning:

Bit 0-15	PEAK Adapter specific (Taken from the **CAN_Status** method).
Bit 16-28	Reserved.
Bit 29	Reserved.
Bit 30	Bus Warning status (repeated from bit 6)
Bit 31	Bus off status (repeat from bit 7)

Platforms: **WIN32**

The driver uses proprietary include files and libraries from PEAK which are not included in the distribution.

vectordrv

This is a device driver for the Vector Informatik (http://www.vector-informatik.com/english/)

DriverString: The driver string has the following format (note that all values can be entered in either decimal or hexadecimal form (for hex precede with 0x).

board;boardidx;channel;filter;mask;bus-speed

board is one of

VIRTUAL or 1 for **virtual adapter**

CANCARDX or 2

CANPARI or 3

CANAC2 or 5

CANAC2PCI or 6

CANCARDY or 12

CANCARDXL or 15 for **PCI Card / PCMCIA**

CANCARD2 or 17

EDICCARD or 19

CANCASEXL or 21 for **USB adapter**

CANBOARDXL or 25

CANBOARDXL_COMPACT or 27

board index

index for board if more then one of same type (0...n)

channel

Is a value from 0 an up indicating the CAN channel on the selected board.

filter

Is the hardware dependent filter for this board hardware. Note that this filter may work in a different way then the CANAL filter.

mask

Is the hardware dependent mask for this board hardware. Note that this filter may work in a different way then the CANAL filter.

bus_speed

The actual bit rate is set here i.e. 125b is given as 125,000

Flags: **bit 0**

        Reserved

bit 1

        reserved

bit 2

0	Filter our own TX messages from our receive..
1	All sent CAN objects appear as received CAN objects in the rx queues.

bit 3

0	Active Mode.
1	Passive mode: CAN controller works as passive CAN node (only monitoring) and therefore it does not send any acknowledge bit for CAN objects sent by another CAN node.

bit 4

0	No error report objects.
1	Error frames are detected and reported as CAN objects via the rx queues

Status return: The CanalGetStatus call returns the status structure with the *channel_status* member having the following meaning:

Bit 0-7	TX error counter
Bit 8-15	RX error counter
Bit 16-17	Reserved
Bit 18	All sent CAN objects appear as received CAN objects.
Bit 19	Passive mode
Bit 20-27	Reserved.
Bit 28	Active.
Bit 29	Bus Passive.
Bit 30	Bus Warning status.
Bit 31	Bus off status.

Platforms: **WIN32**

The driver uses proprietary include files and libraries from Vector which are not included in the distribution.

zanthicdrv

CANAL Driver: zanthicdrv.dll(win32)

Include library for msvc: zanthicdrv.lib

This is a device driver for the Zanthic Technologies (http://www.zanthic.com/products.htm) USB adapter.

DriverString: The driver string has the following format (note that all values can be entered in either decimal or hexadecimal form (for hex preceded with 0x).

                    bus-speed

                    bus_speed This is the speed for the CAN bus. It can be given as

10 for 10 Kbps

20 for 20 Kbps

50 for 50 Kbps

100 for 100 Kbps

125 for 125 Kbps

250 for 250 Kbps

500 for 500 Kbps

800 for 800 Kbps

1000 for 1000 Mobs

Flags: Not used set to 0.

Status return: The CanalGetStatus call returns the status structure with the *channel_status* member having the following meaning:

Bit 0-31   Reserved.

Platforms: **WIN32**

Take it for a test drive

Instructions for windows users

1. Install the canald package. This is done by executing the installation file **setup4canal.exe.**
2. During the installation select the drivers you want to have installed.
3. If you decided to install the **canallogger** driver, confirm the path to the configuration file in the registry (*HKEY_LOCAL_MACHINE\SOFTWARE\canal\canald\driver0*). Edit if necessary.
4. Also check other driver entries in the registry.
5. Start the daemon. This is done by starting the **canalservice** service. This can be done in the control panel under the administrative tools applet and selecting computer management and services and applications or manually start the **canald** application. Canald can also be started from the control panel applet **canalctrl** that can be found in the control panel.

You can now use **CanalDiagnostic** to test canal. This application is in an early development stage but is still useful to some extent. Open the application and bring up a new window by selecting **new** under the file menu. Select the **canal daemon interface** in the dialog box that comes up. This is the interface to the daemon. Other entries are available for other drivers. You can choose one of them to directly connect to a driver. In this case the driver must not be used by the daemon.

Send a couple of messages. If you installed the **canallogger** the messages will be available in this file.

Open one instance of the **loggerWnd** application. Send some more messages from the **CanalDiagnostic** application. They will now be received both in the **canallogger** file and in the **loggerWnd** application.

Open more **loggerWnd** applications and the send more messages and see what happens.

Open a new window in **CanalDiagnostic** and send new messages.

By now you probably understand the functionality.

canald configuration

There is not much configuration to do for canald other than adding/removing interfaces to devices. On the windows platform this is done in the registry.
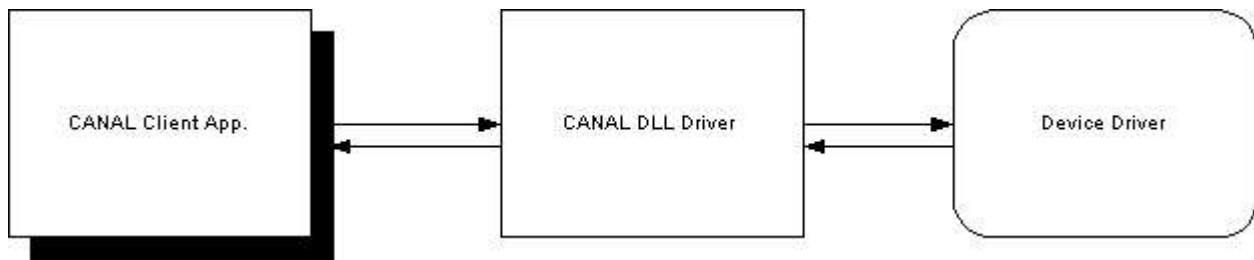
To add a driver.

Locate the driver dll.

Add the driver as a registry entry. Registry entries for canald can be found at **HKEY_LOCAL_MACHINE\SOFTWARE\canal\canald** Here there can be several entries such as

device1=can232,,d:\winnt\system32\can232.dll,64,64,1
device2=logger,c:\canallog.txt,d:\winnt\system32\canallogger.dll,64,64,1

etc.

each device listed here is loaded and activated when canald is started. Each key is of string format.

The format for the keys are

devicen = name,argument,canal dll, inbuffer-size, outbuffer-size, flags

where:

name is a user supplied name for the device which may be used for grouping of devices in the future.

argument This is a string that contains some driver specific information. The string is sent to the driver when it is opened. There must not be any ',' (commas) in this string, use ';' or some other character to separate different part of the argument..

canal dll is the path to the dll that will interface the device.

inbuffer, **outbuffer** sizes are the buffer sizes allocated by canald internally for the device.

flags are a 32-bit value sent to the driver when it is opened. The meaning and interpretation is driver specific.

Drivers are loaded in the order they are entered in the registry. driver**1**, driver**2**.... driver**n** and so on. The first driver get driver id = 1 next driver id = 2 etc. This id is stored in the **obid** part in the canalmsg structure for each received message.

canald usage

After configuration and execution of canald there is not much one can do with the application other then interfacing to it, getting status from it and shutting it down. Status and shutdown control is available by clicking on the icon in the task bar (usually lower right corner). The **canaldown** application can also be used to shut down the daemon.

Make a canal driver for new hardware

The canal interface is just a dll with a specific interface to the world that acts as a black box against the original driver software. It is of course also possible to build the original driver so that it will have the canal interface. When this is done, which is a very easy task, the device is usable by all software that adopts the canal specification. The canal daemon is important software in this case.

A specific piece of software can work against the canal daemon and thus all drivers the daemon serves or just add one specific driver. The program just uses the correct dll.

If you work on the Windows platform there is source in the generic dll that can be used as a template for a new driver. If you are making something that is close in functionality to one of the other available drivers then the one that is closest is the natural start for a new project.

The interface specification

 The full interface specification is available in the **canal_spec.pdf** document which can be downloaded from http://www.vscp.org   This is just a brief description:

open

Open a CAN channel on the device. If the device only has one CAN interface there is only one channel to open. Static information should be set at this state. This could be filters/masks, baud rate and other driver parameters.

close

Close a channel on a device.

send

Send a CAN frame on the channel.

receive

Receive a CAN frame on the channel.

dataAvailable

Check if there is data to receive.

setFilter/setMask

One filter and one mask can be set for the channel. If more filters or masks are needed they have to be set in the open method.

This is it. The rest is method information and some statistics. The interface is therefore very easy to implement. The original interface can be kept for more advanced usage.
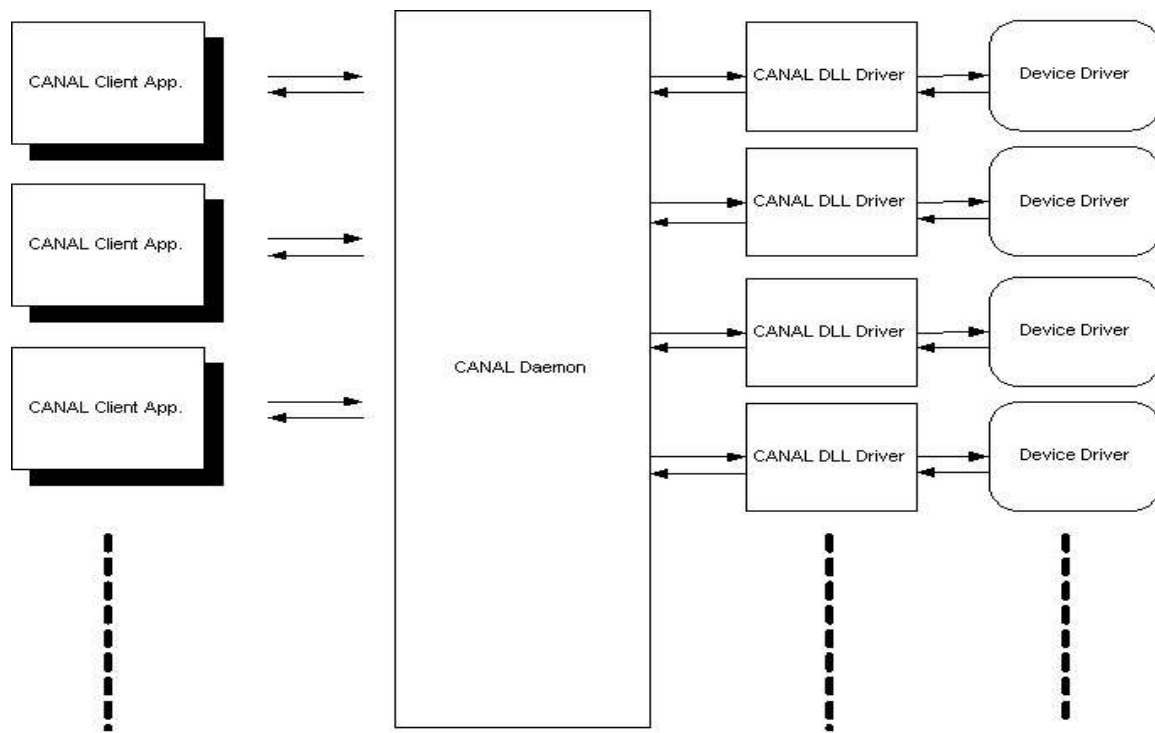
Canald architecture

The canald daemon has four outer interfaces.

1. One is against canal-drivers and is actually several interfaces.

2. One for each driver.

3. One for VSCP Level II clients.

4. One for UDP VSCP Level II datagram receive/send.

Several drivers can be loaded at the same time and when a user sends a message it will be sent to all of them. The drivers must not be just standard CAN devices. With canald comes RS-232 and a TCP/IP implementation. Perfect for simulation where more than one machine is involved.

The other interface is to the application software. This interface is also in the form of a canal interface and works in the same manner as a driver. The exception is that communication now goes through the daemon. In the same way as several drivers against hardware can be loaded several clients can talk to/connect to canald.

So, a message sent from one client will be sent to all other clients and all devices attached to canald.

This can be used for simulation of real world situations. One can add piece after piece to the system and replace simulated devices with "real" devices one by one getting a well controlled migration of a project.

It can also be used for redundant systems where more then one path is needed to certify availability.

Instructions for Linux users

The **canal daemon**, **cancmd** and the **logger driver** has been ported to Linux and is in a stable state. The rest of the applications are being worked on. wxWidgets is used for all applications so porting is trivial in most situations.

In each directory in the source distribution there is a WIN32 folder for the windows version and a Linux folder for Linux versions. In each of these folders is a makefile for that application. Just type make to build it. Our goal is to use autoconf and automake in the future.

To build the system wxWidgets 2.4 is needed. This library can be downloaded from http://www.wxwidgets.org Note that the console versions uses the wxBase version.

On the UNIX platform a configuration file is required, this is /etc/canal.cfg

```
#
# Testfile
#
#device1 = "communication,com1,canal_comm.dll,64,64,1"
#device2 = "simulation,sim1,canal_socket.dll,64,64,1"
device1 =
"logger,/tmp/canallog.txt,/home/akhe/can/src/canal/loggerdll/linux
/canallogger.so,64,64,1"
```

The device entries for each driver in this file are in the same format as for the registry entries for the WIN32 version.