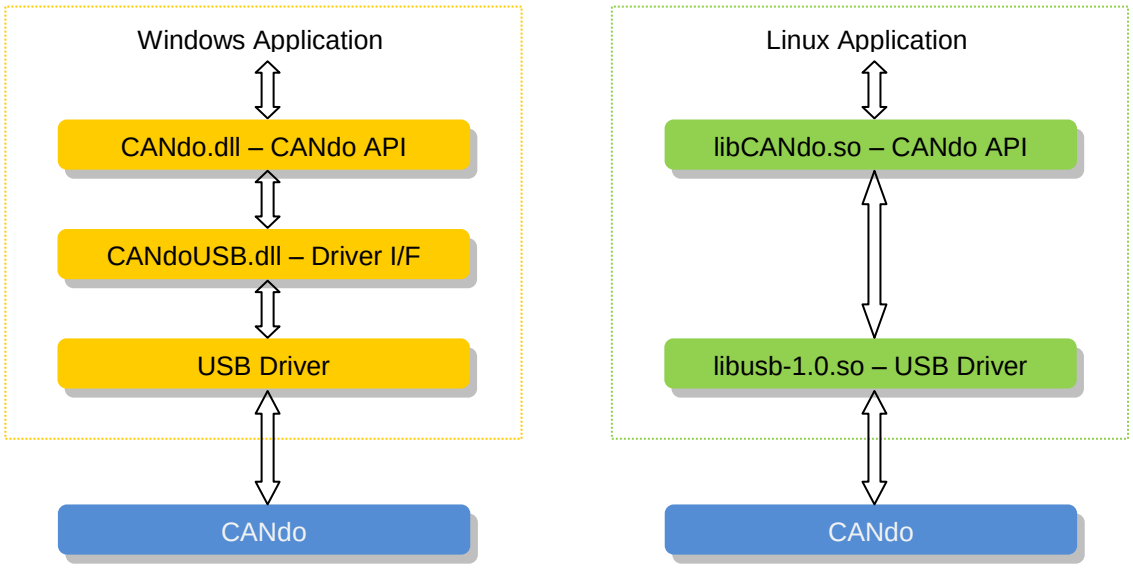
CANdo Programmer's Guide

1 Overview

The CANdo SDK (Software Development Kit) is designed to allow the CANdo, CANdoISO & CANdo AUTO devices to be easily integrated into any Windows or Linux program written in a high level language. Examples are included within the SDK written in C, C#, Pascal & Visual Basic.

On Windows, the core components of the SDK are the CANdo.dll & CANdoUSB.dll dynamic link libraries. The CANdo.dll contains all the API functions necessary to communicate with the CANdo devices, via the CANdoUSB.dll. The CANdoUSB.dll contains the interface between the CANdo.dll & the CANdo driver. Both DLLs must be present in the project directory or within a directory known to Windows, such as the \Windows\System32 or equivalent. The SDK contains both 32 & 64 bit versions of the CANdo DLLs. The CANdo DLLs & the USB driver are designed to operate with Microsoft Windows XP, Vista, 7, 8 & 8.1.

On Linux, the core components of the SDK are the libCANdo.so & libusb-1.0.so shared libraries. The libCANdo.so contains all the API functions necessary to communicate with the CANdo devices, via libusb-1.0.so library. The libusb-1.0.so shared library provides the low level USB interface to the CANdo device. This libusb-1.0 library is not included within the SDK, but is installed by default on most Linux installations. The libCANdo.so library is designed to work with the libusb-1.0 library & the Linux kernel 2.6 or newer.



The CANdo, CANdoISO & CANdo AUTO devices are all software compatible from the perspective of the SDK. However, there are a couple of extra functions supported by only the CANdoISO & CANdo AUTO devices. There are also a group of functions that allow configuration of the CANdo AUTO device via the SDK & these functions are not supported by the other devices. The descriptions of the API functions in the next section describe the common features & the additional features of the CANdoISO & CANdo AUTO devices.

CANdo is the generic name for all the CANdo hardware types & in the descriptions of the API functions below, the name CANdo refers to the CANdo, CANdoISO & CANdo AUTO devices. However, a specific reference to either CANdoISO or CANdo AUTO refers to that device type only.

2 History

Version	Date	Modifications	Author
1.0	01/05/06	Created	MJB
1.1	16/05/06	Appendix B – 'CAN Receive Message Filtering' added	,,
2.0	30/08/10	• Updated for CANdo API DLL v2.0 • CANdoOpen function return values updated • CANdoReceive function return values updated • CANdoGetVersion function added • Appendix C – 'Version Revision Notes' added	,,
2.1	10/01/11	• Updated for CANdo API DLL v2.1 • CANdoClose function modified to return status • CANDO_INVALID_HANDLE return value added	,,
2.2	24/02/11	• Updated for CANdo API DLL v2.2 • CANdoGetDevices function added • CANdoOpenDevice function added • CANdoDevice variable type added • CANdo hardware type constants added • CANdoTransmit function modified	,,
2.3	16/06/11	• Updated for CANdo API DLL v2.3 • CANdoISO support added	,,
3.0	14/11/11	• Updated for CANdo API DLL v3.0 • CANdoStatus NewFlag constants added • CANdoStatus NewFlag operation modified • CANdo USB PID constants added • CANdoGetPID function added • CANdoRequestDateStatus function added • CANdoRequestBusLoadStatus function added • CANdoClearStatus function added • CANdoReceive function modified • CANdoSetBaudRate function updated for CANdoISO	,,
4.0	20/08/13	• Updated for CANdo API DLL v4.0 • CANdoRequestSetupStatus function added • CANdoRequestAnalogInputStatus function added • CANdoAnalogStoreRead function added • CANdoAnalogStoreWrite function added • CANdoAnalogStoreClear function added • CANdoTransmitStoreRead function added • CANdoTransmitStoreWrite function added • CANdoTransmitStoreClear function added	,,
4.1	04/06/14	Linux SDK details added	,,

See Appendix C for details of changes between revisions of the SDK.

3 CANdo API

The CANdo API functions that reside within the API library, CANdo.dll/libCANdo.so, are listed in the section below. The constants & variable types used by the functions are described in Appendix A.

- **CANdoGetPID**(unsigned int CANdoNo, TCANdoDeviceString PID)
- **CANdoGetDevices**(const TCANdoDevices CANdoDevices, unsigned int * NoOfDevices)
- **CANdoOpen**(const PCANdoUSB CANdoUSBPointer)
- **CANdoOpenDevice**(const PCANdoUSB CANdoUSBPointer,
  const PCANdoDevice CANdoDevicePointer)
- **CANdoClose**(const PCANdoUSB CANdoUSBPointer)
- **CANdoFlushBuffers**(const PCANdoUSB CANdoUSBPointer)
- **CANdoSetBaudRate**(const PCANdoUSB CANdoUSBPointer, unsigned char SJW,
  unsigned char BRP, unsigned char PHSEG1, unsigned char PHSEG2,
  unsigned char PROPSEG, unsigned char SAM)
- **CANdoSetMode**(const PCANdoUSB CANdoUSBPointer, unsigned char Mode)
- **CANdoSetFilters**(const PCANdoUSB CANdoUSBPointer,
  unsigned int Rx1Mask,
  unsigned char Rx1IDE1, unsigned int Rx1Filter1,
  unsigned char Rx1IDE2, unsigned int Rx1Filter2,
  unsigned int Rx2Mask,
  unsigned char Rx2IDE1, unsigned int Rx2Filter1,
  unsigned char Rx2IDE2, unsigned int Rx2Filter2,
  unsigned char Rx2IDE3, unsigned int Rx2Filter3,
  unsigned char Rx2IDE4, unsigned int Rx2Filter4)
- **CANdoSetState**(const PCANdoUSB CANdoUSBPointer, unsigned char State)
- **CANdoReceive**(const PCANdoUSB CANdoUSBPointer,
  const PCANdoCANBuffer CANdoCANBufferPointer,
  const PCANdoStatus CANdoStatusPointer)
- **CANdoTransmit**(const PCANdoUSB CANdoUSBPointer,
  unsigned char IDExtended, unsigned int ID,
  unsigned char RTR, unsigned char DLC, const unsigned char * Data,
  unsigned char BufferNo, unsigned char RepeatTime)
- **CANdoRequestStatus**(const PCANdoUSB CANdoUSBPointer)
- **CANdoRequestDateStatus**(const PCANdoUSB CANdoUSBPointer)
- **CANdoRequestBusLoadStatus**(const PCANdoUSB CANdoUSBPointer)
- **CANdoRequestSetupStatus**(const PCANdoUSB CANdoUSBPointer)
- **CANdoRequestAnalogInputStatus**(const PCANdoUSB CANdoUSBPointer)
- **CANdoClearStatus**(const PCANdoUSB CANdoUSBPointer)
- **CANdoGetVersion**(unsigned int * APIVersion, unsigned int * DLLVersion,
  unsigned int * DriverVersion)

The CANdoGetDevices(...) function returns a list of all the CANdo devices connected to the PC, identifying each by hardware type & serial number.

Either the CANdoOpen(…) or CANdoOpenDevice(...) function must be called before using a CANdo device, in order to open a connection to the device. These functions attempt to find a free CANdo on the PC, returning a handle to the device if successful. This handle is part of the TCANdoUSB structure passed into all the other API functions. NOTE : A valid handle must be obtained by calling one of these functions before calling any of the other SDK functions, except for the functions CANdoGetPID(…), CANdoGetDevices(...) & CANdoGetVersion(...), as these don't require a handle. The CANdoOpen(…) function returns a connection to the first free CANdo device found, regardless of hardware type or serial number. The CANdoOpenDevice(…) function allows a particular CANdo device to be selected based on hardware type, serial number or both. A single entry in the CANdoDevices array returned by the CANdoGetDevices(…) function maybe passed into the CANdoOpenDevice(…) function to select a particular device.

After obtaining a handle, the CANdo device is initialised by calls to the CANdoSetBaudRate(…), CANdoSetMode(…) & CANdoSetFilters(…) functions.

Following initialisation, the CANdoSetState(…) function is called to put CANdo into run mode, enabling reception & transmission of CAN messages & starting the internal CAN message timestamp.

The CANdoReceive(…) & CANdoTransmit(…) functions, receive & transmit CAN messages respectively.

The CANdoRequestStatus(…) function requests the internal status of the CANdo unit & the CAN bus. If any errors are detected in the operation of the CANdo unit or an error occurs on the CAN bus, then one or more status messages are automatically generated by the CANdo device & returned to the PC. The status is returned within the TCANdoStatus structure (see Appendix A), which is populated by a call to the CANdoReceive(...) function.

The CANdoRequestDateStatus(…) function requests the date of manufacture of the device. The status is returned in the same manner as for the CANdoRequestStatus(...) function.

The CANdoRequestBusLoadStatus(…) function requests the CAN bus message load on the connected CAN bus. The status is returned in the same manner as for the CANdoRequestStatus(...) function.

The CANdoRequestSetupStatus(...) function requests the CAN baud rate & CAN operating mode of the device (CANdo AUTO device only). The status is returned in the same manner as for the CANdoRequestStatus(...) function.

The CANdoRequestAnalogInputStatus(...) function requests a sample from each of the analogue inputs of the device (CANdo AUTO device only). The status is returned in the same manner as for the CANdoRequestStatus(...) function.

The CANdoClearStatus(…) function clears any pending system status message within the CANdo device. CAN bus related errors are not cleared by this command, as these are handled by the low level CAN controller.

The CANdoGetVersion(…) function returns the versions of the API library, the USB library & the USB driver installed on the PC.

The reception & transmission of CAN messages is stopped by a further call to the CANdoSetState(…) function with the state set to stop.

The CANdoClose(…) function must be called before terminating an application, to close the CANdo connection & free the handle to the device.

The additional API functions listed below also reside within the API library, CANdo.dll/libCANdo.so & are specific to the CANdo AUTO device only & included to permit configuration of the device using the SDK.

- **CANdoAnalogStoreRead**(const PCANdoUSB CANdoUSBPointer)

- **CANdoAnalogStoreWrite**(const PCANdoUSB CANdoUSBPointer,
  unsigned char InputNo, unsigned char IDExtended, unsigned int ID,
  unsigned char Start, unsigned char Length,
  float ScalingFactor, float Offset,
  unsigned char Padding, unsigned char RepeatTime)

- **CANdoAnalogStoreClear**(const PCANdoUSB CANdoUSBPointer)

- **CANdoTransmitStoreRead**(const PCANdoUSB CANdoUSBPointer)

- **CANdoTransmitStoreWrite**(const PCANdoUSB CANdoUSBPointer,
  unsigned char IDExtended, unsigned int ID,
  unsigned char RTR, unsigned char DLC, const unsigned char * Data,
  unsigned char RepeatTime)

- **CANdoTransmitStoreClear**(const PCANdoUSB CANdoUSBPointer)

The CANdoAnalogStoreRead(...), CANdoAnalogStoreWrite(...) & CANdoAnalogStoreClear(...) functions read, write & clear the analogue store in non-volatile memory. The analogue store contains the configuration for the periodic CAN message associated with each analogue input.

The CANdoAnalogStoreClear(...) function erases the configuration within the analogue store, which disables sampling of all the analogue inputs & transmission of the CAN messages associated with the analogue inputs.

The CANdoAnalogStoreWrite(...) function configures the analogue input CAN transmit message & sets the sample/message rate for the input.

The CANdoAnalogStoreRead(...) function requests a read of the contents of the analogue store. The contents are returned within the TCANdoCANBuffer (see Appendix A) structure, which is populated by a call to the CANdoReceive(...) function.

The CANdoTransmitStoreRead(...), CANdoTransmitStoreWrite(...) & CANdoTransmitStoreClear(...) functions read, write & clear the CAN transmit store in non-volatile memory. The CAN transmit store contains the configuration of up to 10 pre-defined fixed, periodic CAN messages.

The CANdoTransmitStoreClear(...) function erases the configuration within the CAN transmit store, which disables the transmission of all the fixed, periodic CAN messages.

The CANdoTransmitStoreWrite(...) function configures the CAN transmit store with a single fixed, periodic CAN transmit message. Each call to the function adds another fixed, periodic message to the store, up to a maximum of 10.

The CANdoTransmitStoreRead(...) function requests a read of the contents of the CAN transmit store. The contents are returned within the TCANdoCANBuffer (see Appendix A) structure, which is populated by a call to the CANdoReceive(...) function.

3.1 CANdoGetPID(...)

Prototype	int CANdoGetPID(unsigned int CANdoNo, TCANdoDeviceString PID)
Parameters	CANdoNo – number of the CANdo device, 0 – n (where n = Total no. of CANdo – 1) PID – the returned USB PID for the specified CANdo device
Return Value	CANDO_SUCCESS CANDO_NOT_FOUND
Description	Returns the USB PID (Product ID) for the CANdo device specified by the CANdoNo parameter. The CANdoNo maybe the No parameter within the TCANdoUSB structure returned by the functions CANdoOpen & CANdoOpenDevice. If only one CANdo device is ever likely to be connected to the PC, then CANdoNo maybe set to 0, to always select the first device. This function is intended to assist in the identification of custom versions of the CANdo device that are programmed with a custom PID.

Prerequisites

CANdo API Library	v3.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	N/A

3.2 CANdoGetDevices(...)

Prototype	int CANdoGetDevices(const TCANdoDevice CANdoDevices[ ], unsigned int * NoOfDevices)
Parameters	CANdoDevices – array of TCANdoDevice NoOfDevices – pointer to a value passed in that specifies the number of devices to enumerate & returns with the number of devices found
Return Value	CANDO_SUCCESS CANDO_USB_DLL_ERROR CANDO_USB_DRIVER_ERROR CANDO_NOT_FOUND
Description	Populates the CANdoDevices array passed into the function with the hardware type & serial number of each device found connected to the PC. The NoOfDevices parameter must be set to the maximum number of devices to search for, before the function is called. Typically, NoOfDevices is set to the declared size of the CANdoDevices array. The value of NoOfDevices must never be greater than the size of the CANdoDevices array. After the function returns, the NoOfDevices parameter contains the number of CANdo devices found.

Prerequisites

CANdo API Library	v2.2 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	N/A

3.3 CANdoOpen(…)

Prototype	int CANdoOpen(const PCANdoUSB CANdoUSBPointer)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure
Return Value	CANDO_SUCCESS CANDO_USB_DLL_ERROR CANDO_USB_DRIVER_ERROR CANDO_NOT_FOUND CANDO_CONNECTION_CLOSED CANDO_IO_FAILED
Description	Finds the next available CANdo connected to the PC. If successful the function returns a handle to the connected device, together with the device description & the serial number.

Prerequisites

CANdo API Library	v1.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	N/A

3.4 CANdoOpenDevice(...)

Prototype	int CANdoOpenDevice(const PCANdoUSB CANdoUSBPointer, const PCANdoDevice CANdoDevicePointer)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure CANdoDevicePointer – pointer to TCANdoDevice structure
Return Value	CANDO_SUCCESS CANDO_USB_DLL_ERROR CANDO_USB_DRIVER_ERROR CANDO_NOT_FOUND CANDO_CONNECTION_CLOSED CANDO_IO_FAILED
Description	Opens a connection to the CANdo device specified by the hardware type & serial number passed into the function via the CANdoDevicePointer parameter. If successful the function returns a handle to the connected device, together with the device description & the serial number, via the CANdoUSBPointer parameter. In addition, the hardware type of the connected device is returned via the CANdoDevicePointer parameter. The CANdoDevice structure passed into the function contains fields for hardware type & serial number. To uniquely specify a CANdo device for connection, both fields must be populated. To select a particular hardware type only, the serial number maybe set to an empty string (“”), the function will then connect to the next free device matching the selected hardware type. To select a device with a specific serial number regardless of hardware type, the hardware type maybe set to CANDO_TYPE_ANY. A call to this function with the serial number set to an empty string & the hardware type set to CANDO_TYPE_ANY is equivalent to a call to the CANdoOpen(…) function, except that the hardware type of the connected device is returned via the CANdoDevicePointer parameter.

Prerequisites

CANdo API Library	v2.2 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	N/A

3.5 CANdoClose(…)

Prototype	int CANdoClose(const PCANdoUSB CANdoUSBPointer)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure
Return Value	CANDO_SUCCESS CANDO_INVALID_HANDLE CANDO_CONNECTION_CLOSED CANDO_ERROR
Description	Closes the connection to the specified CANdo device & frees the handle.

Prerequisites

CANdo API Library	v1.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	N/A

3.6 CANdoFlushBuffers(…)

Prototype	int CANdoFlushBuffers(const PCANdoUSB CANdoUSBPointer)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure
Return Value	CANDO_SUCCESS CANDO_CONNECTION_CLOSED CANDO_IO_FAILED
Description	Flushes the internal USB read & write buffers.

Prerequisites

CANdo API Library	v1.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	N/A

3.7 CANdoSetBaudRate(…)

Prototype

int CANdoSetBaudRate(const PCANdoUSB CANdoUSBPointer, unsigned char SJW, unsigned char BRP, unsigned char PHSEG1, unsigned char PHSEG2, unsigned char PROPSEG, unsigned char SAM)

Parameters

CANdoUSBPointer – pointer to TCANdoUSB structure

SJW – sync. jump width, 0 – 3. (This is the width of the synchronisation jump used by the CAN module to achieve synchronisation. 0 = 1 jump bit … 3 = 4 jump bits)

BRP – baud rate prescaler, 0 – 63 for a CANdo device, 0 – 31 & 64 – 127 for a CANdoISO or CANdo AUTO device (See description below)

PHSEG1 – phase segment 1, 0 – 7 (See description below)

PHSEG2 – phase segment 2, 0 – 7 (See description below)

PROPSEG – propagation segment, 0 – 7 (See description below)

SAM – number of samples per data bit, 0 – 1 (0 = Sample each data bit once, 1 = Sample each data bit thrice)

NOTE : All these CAN register values are 0 based, so that 0 actually equals 1. For example, a PHSEG1 value of 3 equates to a 4 PHSEG1 segments in the CAN bit timing. The equations shown below all expect the 0 based values.

Return Value

CANDO_SUCCESS
CANDO_CONNECTION_CLOSED
CANDO_WRITE_ERROR
CANDO_WRITE_INCOMPLETE
CANDO_ERROR

Description

Sets the CAN bus baud rate (BR) & bit sampling point (SP) according to the following equations -

BR = 20000000 / 2 * (BRP + 1) * (4 +PROPSEG + PHSEG1 + PHSEG2)

NOTE : For a CANdo device, the BRP may be programmed between 0 – 63. For a CANdoISO or CANdo AUTO device the BRP is restricted to between 0 – 31. Within these ranges the BR maybe calculated using the equation shown above. The minimum programmable baud rate for a CANdo device is 6.25k & for a CANdoISO or CANdo AUTO device is 12.5k baud.

SP = (3 + PROPSEG + PHSEG1) * 100 / (4 + PROPSEG + PHSEG1 + PHSEG2)

Some rules apply with respect to the values entered into these equations due to their interdependence upon one another, as described below -

PROPSEG + PHSEG1 + 1 >= PHSEG2
PROPSEG + PHSEG1 + PHSEG2 >= 4
PHSEG2 >= SJW

The table below gives typical settings for some common baud rates -

BR	SP	BRP	PROPSEG	PHSEG1	PHSEG2
50k	70%	9	4	7	5
125k	75%	4	2	7	3
250k	75%	1	5	7	4

500k	70%	0	4	7	5
1M	80%	0	1	4	1

For a CANdoISO or CANdo AUTO device, the baud rate may be programmed with a higher setting resolution using a BRP in the range 64 – 127. In this case the baud rate is calculated using the equation shown below -

BR = 40000000 / 2 * (BRP - 63) * (4 + PROPSEG + PHSEG1 + PHSEG2)

This allows additional baud rates to be programmed that are not available for a CANdo device. For example -

800k = 40000000 / 2 * (64 – 63) * (4 + 7 + 7 + 7)

The sample point calculation remains the same as before.

NOTE : The baud rate settings are stored internally in the CANdo unit in non-volatile memory. This ensures that the unit powers up with the last programmed baud rate settings automatically. The internal memory is only updated if the baud rate parameters specified in the CANdoSetBaudRate(…) function differ from those already stored. If the baud rate parameters do differ, then the unit can take up to 100ms to store the new settings. During this period the unit is unable to accept any new commands.

NOTE : The values used for BRP, PROPSEG, PHSEG1 & PHSEG2 in the above equations for the SDK are all 1 less than the corresponding values shown in the CANdo Application ‘CAN Setup’ page. The values shown in the ‘CAN Setup’ page all start at 1 rather than 0, as per the SDK.

Prerequisites

CANdo API Library	v1.0 or greater
CANdo Firmware	v1.0 or greater
CANdoISO Firmware	v3.0 or greater
CANdo AUTO Firmware	v1.0 or greater

3.8 CANdoSetMode(…)

Prototype	int CANdoSetMode(const PCANdoUSB CANdoUSBPointer, unsigned char Mode)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure Mode – operating mode, 0 – 2 (0 = Normal, 1 = Listen only, 2 = Loopback)
Return Value	CANDO_SUCCESS CANDO_CONNECTION_CLOSED CANDO_WRITE_ERROR CANDO_WRITE_INCOMPLETE CANDO_ERROR
Description	Sets the CAN operating mode. Normal mode – CANdo acts as a normal active CAN node, allowing both reception & transmission of CAN messages. An acknowledgement is automatically sent on successfully receiving a message. Listen only mode – reception of messages enabled only. No acknowledgement is sent on successfully receiving a message. Loopback mode – transmitted messages are routed back to the receiver for testing purposes. No messages are actually transmitted or received on the CAN bus in this mode. NOTE : The operating mode is stored internally in the CANdo unit in non-volatile memory. This ensures that the unit powers up with the last programmed operating mode automatically. The internal memory is only updated if the operating mode parameter specified in the CANdoSetMode(…) function differs from the one already stored. If the operating mode does differ, then the unit can take up to 10ms to store the new setting. During this period the unit is unable to accept any new commands.

Prerequisites

CANdo API Library	v1.0 or greater
CANdo Firmware	v1.0 or greater
CANdoISO Firmware	v1.0 or greater
CANdo AUTO Firmware	v1.0 or greater

3.9 CANdoSetFilters(…)

Prototype	int CANdoSetFilters(const PCANdoUSB CANdoUSBPointer, unsigned int Rx1Mask, unsigned char Rx1IDE1, unsigned int Rx1Filter1, unsigned char Rx1IDE2, unsigned int Rx1Filter2, unsigned int Rx2Mask, unsigned char Rx2IDE1, unsigned int Rx2Filter1, unsigned char Rx2IDE2, unsigned int Rx2Filter2, unsigned char Rx2IDE3, unsigned int Rx2Filter3, unsigned char Rx2IDE4, unsigned int Rx2Filter4)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure Rx1Mask – receive buffer 1 mask Rx1IDE1 – flag to indicate Rx1Mask & Rx1Filter1 values are either 11 or 29 bit values (0 = 11 bit ID, 1 = 29 bit ID) Rx1Filter1 – receive buffer 1, filter 1 (See Appendix B for further details) Rx1IDE2 – flag to indicate Rx1Filter2 value is either an 11 or 29 bit value (0 = 11 bit ID, 1 = 29 bit ID) Rx1Filter2 – receive buffer 1, filter 2 (See Appendix B for further details) Rx2Mask – receive buffer 2 mask Rx2IDE1 – flag to indicate Rx2Mask & Rx2Filter1 values are either 11 or 29 bit values (0 = 11 bit ID, 1 = 29 bit ID) Rx2Filter1 – receive buffer 2, filter 1 (See Appendix B for further details) Rx2IDE2 – flag to indicate Rx2Filter2 value is either an 11 or 29 bit value (0 = 11 bit ID, 1 = 29 bit ID) Rx2Filter2 – receive buffer 2, filter 2 (See Appendix B for further details) Rx2IDE3 – flag to indicate Rx2Filter3 value is either an 11 or 29 bit value (0 = 11 bit ID, 1 = 29 bit ID) Rx2Filter3 – receive buffer 2, filter 3 (See Appendix B for further details) Rx2IDE4 – flag to indicate Rx2Filter4 value is either an 11 or 29 bit value (0 = 11 bit ID, 1 = 29 bit ID) Rx2Filter4 – receive buffer 2, filter 4 (See Appendix B for further details)
Return Value	CANDO_SUCCESS CANDO_CONNECTION_CLOSED CANDO_WRITE_ERROR CANDO_WRITE_INCOMPLETE CANDO_ERROR
Description	Sets the CAN message receive acceptance filters. NOTE : Allow 10ms after sending this command for the CAN module filters to be configured. (See Appendix B for further details).
Prerequisites	CANdo API Library – v1.0 or greater

	CANdo Firmware	v1.0 or greater
	CANdoISO Firmware	v1.0 or greater
	CANdo AUTO Firmware	v1.0 or greater

3.10 CANdoSetState(…)

Prototype	int CANdoSetState(const PCANdoUSB CANdoUSBPointer, unsigned char State)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure State – run state, 0 - 1 (0 = Stop, 1 = Run)
Return Value	CANDO_SUCCESS CANDO_CONNECTION_CLOSED CANDO_WRITE_ERROR CANDO_WRITE_INCOMPLETE CANDO_ERROR
Description	Sets the run state. Stop – disables reception & transmission of messages on the CAN bus. Also disables the CAN transceiver. Run – enables transmission & reception of CAN messages. Also resets the message timestamp & enables the CAN transceiver.

Prerequisites

CANdo API Library	v1.0 or greater
CANdo Firmware	v1.0 or greater
CANdoISO Firmware	v1.0 or greater
CANdo AUTO Firmware	v1.0 or greater

3.11 CANdoReceive(…)

Prototype

int CANdoReceive(const PCANdoUSB CANdoUSBPointer,
const PCANdoCANBuffer CANdoCANBufferPointer,
const PCANdoStatus CANdoStatusPointer)

Parameters

CANdoUSBPointer – pointer to TCANdoUSB structure

CANdoCANBufferPointer – pointer to TCANdoCANBuffer structure

CANdoStatusPointer – pointer to TCANdoStatus structure

Return Value

CANDO_SUCCESS
CANDO_CONNECTION_CLOSED
CANDO_READ_ERROR
CANDO_BUFFER_OVERFLOW
CANDO_RX_OVERRUN
CANDO_RX_TYPE_UNKNOWN
CANDO_RX_CRC_ERROR
CANDO_RX_DECODE_ERROR

Description

Receives CAN & status messages sent by CANdo.

The CAN receive message cyclic buffer is passed into the function via the CANdoCANBufferPointer parameter. The CANdoReceive(…) function populates this cyclic buffer with any CAN messages received via the CAN bus.

A single status message is returned by the CANdoReceive(…) function if an error occurs or a specific status request is sent using one of the functions, CANdoRequestStatus(…), CANdoRequestDateStatus(…), CANdoRequestBusLoadStatus(…) or CANdoRequestSetupStatus(...). The content of the status message depends upon the type of status message returned by the device & is described in detail in Appendix A, under the TCANdoStatus type description.

Prerequisites

CANdo API Library	v1.0 or greater
CANdo Firmware	v1.0 or greater
CANdoISO Firmware	v1.0 or greater
CANdo AUTO Firmware	v1.0 or greater

3.12 CANdoTransmit(…)

Prototype

int CANdoTransmit(const PCANdoUSB CANdoUSBPointer,
unsigned char IDExtended, unsigned int ID,
unsigned char RTR, unsigned char DLC, const unsigned char * Data,
unsigned char BufferNo, unsigned char RepeatTime)

Parameters

CANdoUSBPointer – pointer to TCANdoUSB structure

IDExtended – flag to indicate 11 or 29 bit message ID, 0 – 1 (0 = 11 bit ID, 1 = 29 bit ID)

ID – CAN message ID, 0 – 7FF (11 bit ID) or 0 – 1FFFFFFF (29 bit ID)

RTR – remote frame flag, 0 – 1 (0 = Data, 1 = Remote frame)

DLC – Data Length Code, 0 – 8 (No. of data bytes in message)

Data – byte array containing data to be transmitted (Note : A null pointer is permitted for a remote frame transmission, see note below)

Buffer No – buffer within CANdo to use for the transmission, 0 - 15. Buffer 0 transmits directly to the CAN bus. Buffers 1 – 15 are specialised repeat buffers that allow accurate timed transmissions of repetitive messages. To transmit a message at a specified rate, select one of the buffers 1 – 15 & set the RepeatTime parameter to one of the values specified below. Once loaded, a repeat buffer transmits the message in the buffer repetitively, at the rate specified by the RepeatTime parameter. To stop a repeat buffer transmitting, set the RepeatTime to 0

RepeatTime – the message transmit repeat time, 0 – 10. (Only applies to the repeat buffers 1 – 15, set to 0 for buffer 0)

0	Off
1	10ms
2	20ms
3	50ms
4	100ms
5	200ms
6	500ms
7	1000ms
8	2000ms
9	5000ms
10	10000ms

Return Value

CANDO_SUCCESS
CANDO_CONNECTION_CLOSED
CANDO_WRITE_ERROR
CANDO_WRITE_INCOMPLETE
CANDO_ERROR

Description

Transmits a message on the CAN bus.

The transmitter includes sixteen buffers (0 – 15), one single shot (buffer 0) & fifteen (buffers 1 – 15) repeat buffers. A message loaded into buffer 0 is transmitted once only. The repeat buffers once loaded with a CAN message, repeat the transmission at the rate specified by the RepeatTime parameter. The repeat buffers allow for accurate repetitive transmissions on the CAN bus with no overhead on the PC.

NOTE : v2.2 & greater of the CANdo API library allows the ‘Data’ pointer to be null. This is intended for use when transmitting a remote frame that doesn’t require data.

Prerequisites

CANdo API Library	v1.0 or greater
CANdo Firmware	v1.0 or greater
CANdoISO Firmware	v1.0 or greater
CANdo AUTO Firmware	v1.0 or greater

3.13 CANdoRequestStatus(…)

Prototype	int CANdoRequestStatus(const PCANdoUSB CANdoUSBPointer)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure
Return Value	CANDO_SUCCESS CANDO_CONNECTION_CLOSED CANDO_WRITE_ERROR CANDO_WRITE_INCOMPLETE CANDO_ERROR
Description	Requests the CAN bus & internal CANdo status. After sending this command to CANdo, the status is transmitted back to the PC in <1ms. To read the status, call the CANdoReceive(…) function & interrogate the NewFlag & status information within the TCANdoStatus parameter. (A status message is automatically sent back to the PC if an error is detected on the CAN bus, or if there is an internal system error within the CANdo device.)

Prerequisites

CANdo API Library	v1.0 or greater
CANdo Firmware	v1.0 or greater
CANdoISO Firmware	v1.0 or greater
CANdo AUTO Firmware	v1.0 or greater

3.14 CANdoRequestDateStatus(…)

Prototype	int CANdoRequestDateStatus(const PCANdoUSB CANdoUSBPointer)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure
Return Value	CANDO_SUCCESS CANDO_CONNECTION_CLOSED CANDO_WRITE_ERROR CANDO_WRITE_INCOMPLETE CANDO_ERROR
Description	Requests the date of manufacture of the CANdo device. After sending this command to CANdo, the date status is transmitted back to the PC in <1ms. To read the status, call the CANdoReceive(…) function & interrogate the NewFlag & status information within the TCANdoStatus parameter.

Prerequisites

CANdo API Library	v3.0 or greater
CANdo Firmware	v3.0 or greater
CANdoISO Firmware	v3.0 or greater
CANdo AUTO Firmware	v1.0 or greater

3.15 CANdoRequestBusLoadStatus(…)

Prototype	int CANdoRequestBusLoadStatus(const PCANdoUSB CANdoUSBPointer)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure
Return Value	CANDO_SUCCESS CANDO_CONNECTION_CLOSED CANDO_WRITE_ERROR CANDO_WRITE_INCOMPLETE CANDO_ERROR
Description	Requests the CAN bus loading as calculated by the CANdoISO or CANdo AUTO device. The bus load is calculated every second, while the device is running. After sending this command to the device, the bus load status is transmitted back to the PC in <1ms. To read the status, call the CANdoReceive(…) function & interrogate the NewFlag & status information within the TCANdoStatus parameter. NOTE : This function is supported by the CANdoISO & CANdo AUTO devices only. The command is ignored by the CANdo device.

Prerequisites

CANdo API Library	v3.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	v3.0 or greater
CANdo AUTO Firmware	v1.0 or greater

3.16 CANdoRequestSetupStatus(...)

Prototype	int CANdoRequestSetupStatus(const PCANdoUSB CANdoUSBPointer)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure
Return Value	CANDO_SUCCESS CANDO_CONNECTION_CLOSED CANDO_WRITE_ERROR CANDO_WRITE_INCOMPLETE CANDO_ERROR
Description	Requests the setup status of the connected device, this includes the CAN baud rate & the operating mode. After sending this command to the device, the setup status is transmitted back to the PC in <1ms. To read the status, call the CANdoReceive(…) function & interrogate the NewFlag & status information within the TCANdoStatus parameter. NOTE : This function is supported by the CANdo AUTO device only. The command is ignored by the CANdo & CANdoISO devices.

Prerequisites

CANdo API Library	v4.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A

	CANdo AUTO Firmware	v1.0 or greater

3.17 CANdoRequestAnalogInputStatus

Prototype	int CANdoRequestAnalogInputStatus(const PCANdoUSB CANdoUSBPointer)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure
Return Value	CANDO_SUCCESS CANDO_CONNECTION_CLOSED CANDO_WRITE_ERROR CANDO_WRITE_INCOMPLETE CANDO_ERROR
Description	Requests the status of the analogue inputs of the connected CANdo AUTO device. After sending this command to the device, the device samples & digitises each analogue input & then transmits the measured values back to the PC in <1ms. To read the status, call the CANdoReceive(…) function & interrogate the NewFlag & status information within the TCANdoStatus parameter. NOTE : This function is supported by the CANdo AUTO device only. The command is ignored by the CANdo & CANdoISO devices.

Prerequisites

CANdo API Library	v4.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	v1.0 or greater

3.18 CANdoClearStatus(…)

Prototype	int CANdoClearStatus(const PCANdoUSB CANdoUSBPointer)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure
Return Value	CANDO_SUCCESS CANDO_CONNECTION_CLOSED CANDO_WRITE_ERROR CANDO_WRITE_INCOMPLETE CANDO_ERROR
Description	Clears the internal system status within the CANdo device. The CAN bus status is not cleared by this function, as this is determined by the state of the CAN bus & CAN module only.

Prerequisites

CANdo API Library	v3.0 or greater
CANdo Firmware	v3.0 or greater
CANdoISO Firmware	v3.0 or greater
CANdo AUTO Firmware	v1.0 or greater

3.19 CANdoGetVersion(…)

Prototype	void CANdoGetVersion(unsigned int * APIVersion, unsigned int * DLLVersion, unsigned int * DriverVersion)
Parameters	APIVersion – pointer that returns the API library version x 10 DLLVersion – pointer that returns the USB library version x 10 DriverVersion – pointer that returns the USB driver version x 10
Return Value	None.
Description	Returns the versions of the API & USB libraries & the USB driver. The values returned by this function are the version numbers multiplied by 10. To extract the actual version number, divide the returned value by 10 as a float value, eg. 20 / 10 = v2.0.

Prerequisites

CANdo API Library	v2.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	N/A

3.20 CANdoAnalogStoreClear(...)

Prototype	int CANdoAnalogStoreClear(const PCANdoUSB CANdoUSBPointer)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure
Return Value	CANDO_SUCCESS CANDO_CONNECTION_CLOSED CANDO_WRITE_ERROR CANDO_WRITE_INCOMPLETE CANDO_ERROR
Description	Erases the configuration within the analogue input store. A call to this function disables the sampling of the analogue inputs & stops all associated CAN transmit messages. NOTE : This function is supported by the CANdo AUTO device only. The command is ignored by the CANdo & CANdoISO devices.

Prerequisites

CANdo API Library	v4.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	v1.0 or greater

3.21 CANdoAnalogStoreWrite(...)

Prototype	int CANdoAnalogStoreWrite(const PCANdoUSB CANdoUSBPointer, unsigned char InputNo, unsigned char IDExtended, unsigned int ID, unsigned char Start, unsigned char Length, float ScalingFactor, float Offset, unsigned char Padding, unsigned char RepeatTime)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure InputNo – analogue input no., 1 – 2 (1 = V1 Input, 2 = V2 Input) IDExtended – flag to indicate 11 or 29 bit message ID, 0 – 1 (0 = 11 bit ID, 1 = 29 bit ID) ID – CAN message ID, 0 – 7FF (11 bit ID) or 0 – 1FFFFFFF (29 bit ID) Start – start byte no. of sampled analogue data in data portion of message, 1 – 8 Length – length in bytes of sampled analogue data in data portion of CAN message, 1 – 4 ScalingFactor – multiplication factor applied to sampled analogue input data before insertion into data portion of CAN message Offset – value added to sampled analogue input data before insertion into data portion of CAN message Padding – value of unused data bytes in CAN message RepeatTime – the message transmit repeat time, 0 – 10.

0	Off
1	10ms
2	20ms
3	50ms
4	100ms
5	200ms
6	500ms
7	1000ms
8	2000ms
9	5000ms
10	10000ms

Return Value

CANDO_SUCCESS
CANDO_CONNECTION_CLOSED
CANDO_WRITE_ERROR
CANDO_WRITE_INCOMPLETE
CANDO_ERROR

Description

Writes the configuration of the CAN message for the specified InputNo to the analogue input store, in non-volatile memory.

The sampled analogue input data is scaled & offset & then inserted into the data portion of the CAN message. The data is inserted into the CAN data starting at the Start byte no. & occupying Length bytes. The sampled input data is scaled & offset according to the equation given below.

CAN Data Value = (Vn Input sampled value in mV * ScalingFactor) + Offset

NOTE : This function is supported by the CANdo AUTO device only. The command is ignored by the CANdo & CANdoISO devices.

Prerequisites

CANdo API Library	v4.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	v1.0 or greater

3.22 CANdoAnalogStoreRead(...)

Prototype int CANdoAnalogStoreRead(const PCANdoUSB CANdoUSBPointer)

Parameters CANdoUSBPointer – pointer to TCANdoUSB structure

Return Value
CANDO_SUCCESS
CANDO_CONNECTION_CLOSED
CANDO_WRITE_ERROR
CANDO_WRITE_INCOMPLETE
CANDO_ERROR

Description Requests the contents of the analogue input configuration store.

After sending this command to the device, the analogue input store contents are sent back to the PC in <1ms. To retrieve the contents, call the CANdoReceive(…) function. The TCANdoCANBuffer parameter passed into the CANdoReceive(...) function then holds the configuration data for the store. Each entry in the buffer holds the configuration data for one input within a TCANdoCAN structure. The first entry in the buffer pertains to the V1 Input & the second the V2 Input. The content of the TCANdoCAN structure is detailed in the table below.

TCANdoCAN Parameter	Analogue Input Store Parameter
IDE	IDE
RTR	0
ID	ID
DLC	8
Data[0]	Start
Data[1]	Length
Data[2] to Data[4]	ScalingFactor
Data[5] to Data[7]	Offset
BusState	Padding
Timestamp	RepeatTime

NOTE : This function is supported by the CANdo AUTO device only. The command is ignored by the CANdo & CANdoISO devices.

Prerequisites

CANdo API Library	v4.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	v1.0 or greater

3.23 CANdoTransmitStoreClear(...)

Prototype

int CANdoTransmitStoreClear(const PCANdoUSB CANdoUSBPointer)

Parameters

CANdoUSBPointer – pointer to TCANdoUSB structure

Return Value

CANDO_SUCCESS
CANDO_CONNECTION_CLOSED
CANDO_WRITE_ERROR
CANDO_WRITE_INCOMPLETE
CANDO_ERROR

Description

Erases the configuration within the CAN transmit store.

A call to this function clears the CAN transmit store configuration & disables all the fixed, periodic CAN transmit messages.

NOTE : This function is supported by the CANdo AUTO device only. The command is ignored by the CANdo & CANdoISO devices.

Prerequisites

CANdo API Library	v4.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	v1.0 or greater

3.24 CANdoTransmitStoreWrite(...)

Prototype

int CANdoTransmitStoreWrite(const PCANdoUSB CANdoUSBPointer,
unsigned char IDExtended, unsigned int ID,
unsigned char RTR, unsigned char DLC,
const unsigned char * Data, unsigned char RepeatTime)

Parameters

CANdoUSBPointer – pointer to TCANdoUSB structure

IDExtended – flag to indicate 11 or 29 bit message ID, 0 – 1 (0 = 11 bit ID, 1 = 29 bit ID)

ID – CAN message ID, 0 – 7FF (11 bit ID) or 0 – 1FFFFFFF (29 bit ID)

RTR – remote frame flag, 0 – 1 (0 = Data, 1 = Remote frame)

DLC – Data Length Code, 0 – 8 (No. of data bytes in message)

Data – byte array containing data to be transmitted (Note : A null pointer is permitted for a remote frame transmission)

RepeatTime – the repeat time of the input sampling & CAN message transmission, 0 - 10

0	Off
1	10ms
2	20ms
3	50ms
4	100ms
5	200ms
6	500ms
7	1000ms

	8 2000ms 9 5000ms 10 10000ms
Return Value	CANDO_SUCCESS CANDO_CONNECTION_CLOSED CANDO_WRITE_ERROR CANDO_WRITE_INCOMPLETE CANDO_ERROR
Description	Writes the configuration of a single fixed, periodic CAN message to the CAN transmit store, in non-volatile memory. Up to 10 CAN transmit messages may be written to the CANdo AUTO device by repeated calls to this function. NOTE : This function is supported by the CANdo AUTO device only. The command is ignored by the CANdo & CANdoISO devices.

Prerequisites

CANdo API Library	v4.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	v1.0 or greater

3.25 CANdoTransmitStoreRead(...)

Prototype	int CANdoTransmitStoreRead(const PCANdoUSB CANdoUSBPointer)
Parameters	CANdoUSBPointer – pointer to TCANdoUSB structure
Return Value	CANDO_SUCCESS CANDO_CONNECTION_CLOSED CANDO_WRITE_ERROR CANDO_WRITE_INCOMPLETE CANDO_ERROR
Description	Requests the contents of the CAN transmit configuration store. After sending this command to the device, the CAN transmit store contents are sent back to the PC in <1ms. To retrieve the contents, call the CANdoReceive(…) function. The TCANdoCANBuffer parameter passed into the CANdoReceive(...) function then holds the configuration data for the store. Each entry in the buffer holds the configuration data for one CAN transmit message within the TCANdoCAN structure. NOTE : This function is supported by the CANdo AUTO device only. The command is ignored by the CANdo & CANdoISO devices.

Prerequisites

CANdo API Library	v4.0 or greater
CANdo Firmware	N/A
CANdoISO Firmware	N/A
CANdo AUTO Firmware	v1.0 or greater

Appendix A – Constants & Type Definitions

CONSTANTS

Parameter	Value	Description
CANDO_CLOSED	0x00	No connection to CANdo
CANDO_OPEN	0x01	Connection to CANdo open
CANDO_STOP	0x00	Stop Rx/Tx of CAN messages
CANDO_RUN	0x01	Start Rx/Tx of CAN messages
CANDO_NORMAL_MODE	0x00	Normal Rx/Tx CAN mode
CANDO_LISTEN_ONLY_MODE	0x01	Rx only mode, no ACKs
CANDO_LOOPBACK_MODE	0x02	Tx internally looped back to Rx
CANDO_ID_11_BIT	0x00	Standard 11 bit ID
CANDO_ID_29_BIT	0x01	Extended 29 bit ID
CANDO_DATA_FRAME	0x00	CAN data frame
CANDO_REMOTE_FRAME	0x01	CAN remote frame
CANDO_TYPE_ANY	0x0000	Any CANdo hardware type
CANDO_TYPE_CANDO	0x0001	CANdo hardware type
CANDO_TYPE_CANDOISO	0x0002	CANdoISO hardware type
CANDO_TYPE_CANDO_AUTO	0x0003	CANdo AUTO hardware type
CANDO_TYPE_UNKNOWN	0x8000	CANdo hardware type unknown
CANDO_NO_STATUS	0x00	No status available
CANDO_DEVICE_STATUS	0x01	Device status available
CANDO_DATE_STATUS	0x02	Date status available
CANDO_BUS_LOAD_STATUS	0x03	CAN bus load status available
CANDO_PID	“8095”	CANdo USB PID
CANDOISO_PID	“8660”	CANdoISO USB PID
CANDO_AUTO_PID	“889B”	CANdo AUTO USB PID
CANDO_STRING_LENGTH	0x0100	CANdo string length
CANDO_CAN_BUFFER_LENGTH	0x0800	Size of CAN message receive cyclic buffer
CANDO_CLK_FREQ	20000	CANdo clock frequency for CAN baud rate calculations
CANDO_CLK_FREQ_HIGH	40000	CANdoISO & CANdo AUTO clock frequency for CAN baud rate calculations
CANDO_AUTO_V1_INPUT	0x00	CANdo AUTO V1 analogue input
CANDO_AUTO_V2_INPUT	0x01	CANdo AUTO V2 analogue input
CANDO_AUTO_ANALOG_RESOLUTION	0.0315	CANdo AUTO analogue input resolution (31.5mV)

FUNCTION RETURN CODES

Function Return Code	Value	Description
CANDO_SUCCESS	0x0000	All OK
CANDO_USB_DLL_ERROR	0x0001	Error loading USB library
CANDO_USB_DRIVER_ERROR	0x0002	Error loading USB Driver
CANDO_NOT_FOUND	0x0004	CANdo not found
CANDO_IO_FAILED	0x0008	Failed to initialise USB parameters
CANDO_CONNECTION_CLOSED	0x0010	No CANdo communications channel open
CANDO_READ_ERROR	0x0020	USB read error
CANDO_WRITE_ERROR	0x0040	USB write error
CANDO_WRITE_INCOMPLETE	0x0080	Not all requested bytes written to CANdo
CANDO_BUFFER_OVERFLOW	0x0100	Overflow in cyclic buffer
CANDO_RX_OVERRUN	0x0200	Message received greater than max. message size
CANDO_RX_TYPE_UNKNOWN	0x0400	Unknown message type received
CANDO_RX_CRC_ERROR	0x0800	CRC mismatch
CANDO_RX_DECODE_ERROR	0x1000	Error decoding message
CANDO_INVALID_HANDLE	0x2000	Invalid device handle
CANDO_ERROR	0x8000	Non specific error

NOTE : Function return codes are logically or'ed, so a function may return more than one code.

TYPES

TCANdoDeviceString

```
// Array of characters used to store a CANdo string
typedef unsigned char TCANdoDeviceString[CANDO_STRING_LENGTH];
```

TCANdoDevice

```
// Structure type used to store CANdo identification information
typedef struct TCANdoDevice
{
  int HardwareType;  // Hardware type of this CANdo
  TCANdoDeviceString SerialNo;  // USB S/N for this CANdo
} TCANdoDevice;
```

HardwareType – the CANdo hardware type. The possible values are listed in Appendix A, under constants, of the form 'CANDO_TYPE_*x*'.

SerialNo – the device serial number stored as a null terminated 'C' style string.

TCANdoUSB

```
// Structure type used to store info. relating to connected CANdo
typedef struct TCANdoUSB
{
  int TotalNo;  // Total no. of CANdo on USB bus
  int No;  // No. of this CANdo
  unsigned char OpenFlag;  // USB communications channel state
  TCANdoDeviceString Description;  // USB descriptor string for CANdo
  TCANdoDeviceString SerialNo;  // USB S/N for this CANdo
  HANDLE Handle;  // Handle to connected CANdo
} TCANdoUSB;
```

TotalNo – the total number of CANdo devices connected to the PC.

No – the number of this connected CANdo.

OpenFlag – a flag to indicate that the connected device is open (0 = CLOSED, 1 = OPEN) for communication. Do not attempt to communicate with a device if the connection is closed.

Description – the device description stored as a null terminated 'C' style string.

SerialNo – the device serial number stored as a null terminated 'C' style string.

Handle – a windows handle to the connected device, used in all subsequent communication to identify the device.

TCANdoCAN

```
// Structure type used to store a CAN message
typedef struct TCANdoCAN
{
  unsigned char IDE;
  unsigned char RTR;
  unsigned int ID;
  unsigned char DLC;
  unsigned char Data[8];
  unsigned char BusState;
  unsigned int TimeStamp;
} TCANdoCAN;
```

IDE – flag to indicate length of CAN ID, 0 = 11 bit ID, 1 = 29 bit ID.

RTR – flag to indicate a remote frame, 0 = data, 1 = remote frame.

ID – 11 or 29 bit ID as specified by the IDE flag.

DLC – (Data Length Code) no. of bytes in message, 0 – 8 bytes.

Data – array of data bytes in message.

BusState – CAN bus status.
(See 'CAN Bus Status' table below for further details).

Timestamp – a timestamp indicating the receive time of the message since starting CANdo. The timestamp resolution is 25.6us per bit.

TCANdoCANBuffer

```
// Structure type used as a cyclic buffer to store decoded CAN messages received from CANdo
typedef struct TCANdoCANBuffer
{
  TCANdoCAN CANMessage[CANDO_CAN_BUFFER_LENGTH];
  int WriteIndex;
  int ReadIndex;
  unsigned char FullFlag;
} TCANdoCANBuffer;
```

CANMessage[…] – cyclic buffer for CAN receive messages.
(NOTE : The size of the cyclic buffer is currently fixed within the CANdo.dll/libCANdo.so to the value of CANDO_CAN_BUFFER_LENGTH, this must not be changed.)

WriteIndex – write index pointer for cyclic buffer. Automatically incremented within the CANdo.dll/libCANdo.so as new CAN messages are received.

ReadIndex – read index pointer for cyclic buffer. Increment this index pointer after reading a message in the cyclic buffer.

FullFlag – flag to indicate that the cyclic buffer is full. This occurs when the last free slot in the cyclic buffer is filled with a new message & the write index is incremented so that it equals the read index.

TCANdoStatus

```
// Structure type used to store status information received from CANdo
typedef struct TCANdoStatus
{
  unsigned char HardwareVersion;
  unsigned char SoftwareVersion;
  unsigned char Status;
  unsigned char BusState;
  unsigned int TimeStamp;
  unsigned char NewFlag;
} TCANdoStatus;
```

HardwareVersion – content depends upon the value of the NewFlag parameter (see 'TCANdoStatus Content Versus NewFlag Value' table below for further details) –

1) The version of the CANdo hardware x 10. Divide the HardwareVersion by 10 as a float value to get the actual version number, eg. 21 / 10 = v2.1
2) The CAN bus load whole part as a percentage ranging from 0 – 100%
3) CAN baud rate BRP ranging from 0 - 127
4) V1 Input sample LSB (Least Significant Byte)

SoftwareVersion – content depends upon the value of the NewFlag parameter (see 'TCANdoStatus Content Versus NewFlag Value' table below for further details) –

1) The version of the CANdo software x 10. Divide the SoftwareVersion by 10 as a float value to get the complete version number, eg. 10 / 10 = v1.0
2) Day of manufacture of the device in the range 1 – 31
3) The CAN bus load fractional part as a percentage ranging from 0.0 – 0.9%
4) CAN baud rate PHSEG1 & PHSEG2, bits 0-3 = PHSEG1, bits 4-7 = PHSEG2
5) V1 Input sample MSB (Most Significant Byte)

Status – content depends upon the value of the NewFlag parameter (see 'TCANdoStatus Content Versus NewFlag Value' table below for further details) –

1) Internal status of the CANdo device (see the 'CANdo Status' table below for further details)
2) Month of manufacture of the device in the range 1 – 12
3) CAN module receive error counter
4) CAN baud rate PROPSEG, SJW & SAM, bit 0 = SAM, bit 1 = SJW, bits 4-7 = PROPSEG
5) V2 Input sample LSB (Least Significant Byte)

BusState – content depends upon the value of the NewFlag parameter (see 'TCANdoStatus Content Versus NewFlag Value' table below for further details) –

1) CAN bus status (see 'CAN Bus Status' table below for further details)
2) Year of manufacture of the device in the range 1 – 99 (2001 – 2099)
3) CAN module transmit error counter
4) CAN operating mode (see table below for possible values)
5) V2 Input sample MSB (Most Significant Byte)

BusState - CAN Operating Mode Value	CAN Operating Mode
0x00	Normal
0x20	Sleep
0x40	Loopback
0x60	Listen Only
0x80	Configuration

Timestamp – a timestamp indicating the receive time of the message since starting CANdo. The timestamp resolution is 25.6us per bit.

NewFlag – a flag to indicate the arrival of a new status message. Clear this flag after reading the status message.

Prior to v3.0 of the API library, the NewFlag took one of two values, 0 indicated a FALSE state & any other value indicated a TRUE state, ie. a new status message present. From v3.0 of the API library onwards, the NewFlag may take one of several values that indicate the type of status message

present in the TCANdoStatus parameters. The NewFlag may take on any of the values as defined by the constants –

CANDO_NO_STATUS
CANDO_DEVICE_STATUS
CANDO_DATE_STATUS
CANDO_BUS_LOAD_STATUS
CANDO_SETUP_STATUS
CANDO_ANALOG_INPUT_STATUS

The CANDO_NO_STATUS is the equivalent to the old FALSE & indicates the cleared state, ie. no status message present. The CANDO_DEVICE_STATUS is equivalent to the old TRUE, indicating a device status message is present in the TCANdoStatus parameters.

The table on the next page details the content of the TCANdoStatus parameters based upon the value of the NewFlag.

TCANdoStatus Content Versus NewFlag Value

TCANdoStatus Parameter	NewFlag Value				
	CANDO_DEVICE_STATUS	CANDO_DATE_STATUS	CANDO_BUS_LOAD_STATUS	CANDO_SETUP_STATUS	CANDO_ANALOG_INPUT_STATUS
HardwareVersion	Hardware version	Hardware version	CAN bus load 0-100%	CAN baud rate BRP	[*1] V1 Input sample LSB
SoftwareVersion	Software version	Day of manufacture	CAN bus load 0.0-0.9%	CAN baud rate PHSEG1 & PHSEG2	[*1] V1 Input sample MSB
Status	System status	Month of manufacture	CAN receive error count	CAN baud rate PROPSEG, SJW & SAM	[*1] V2 Input sample LSB
BusState	Bus state	Year of manufacture	CAN transmit error count	CAN operating mode	[*1] V2 Input sample MSB
Timestamp	Timestamp	Timestamp	Timestamp	Timestamp	Timestamp

[*1] Vn Value = Vn Input sample LSB + (Vn Input sample MSB * 256)

CANdo Status

Constant	Value	Description
CANDO_OK	0x00	All OK
CANDO_USB_RX_OVERRUN	0x01	USB port receive message overrun
CANDO_USB_RX_CORRUPTED	0x02	USB port receive message invalid
CANDO_USB_RX_CRC_ERROR	0x04	USB port receive message CRC error
CANDO_CAN_RX_NO_DATA	0x08	CAN receive message no data
CANDO_CAN_RX_OVERRUN	0x10	CAN receive message overrun
CANDO_CAN_RX_INVALID	0x20	CAN receive message invalid
CANDO_CAN_TX_OVERRUN	0x40	CAN transmit message overrun
CANDO_CAN_BUS_ERROR	0x80	CAN bus error

NOTE : These status codes maybe logically or'ed

CAN Bus Status

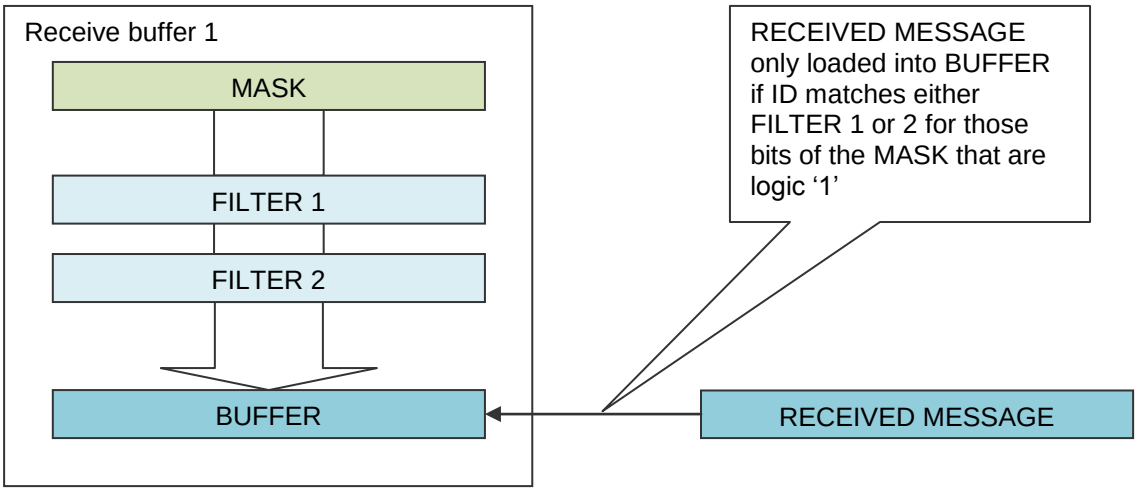
Constant	Value	Description
CAN_OK	0x00	All OK
CAN_WARN	0x01	Rx/Tx warning (>95 errors)
CAN_RX_WARN	0x02	Receiver warning (>95 errors)
CAN_TX_WARN	0x04	Transmitter warning (>95 errors)
CAN_RX_PASSIVE	0x08	Receiver bus passive (>127 errors)
CAN_TX_PASSIVE	0x10	Transmitter bus passive (>127 errors)
CAN_TX_OFF	0x20	Transmitter bus off (>255 errors)
CAN_RX1_OVERFLOW	0x40	Receive buffer 1 overflow
CAN_RX2_OVERFLOW	0x80	Receive buffer 2 overflow

NOTE : These status codes maybe logically or'ed

Appendix B – CAN Receive Message Filtering

CANdo contains two CAN receive message buffers that operate independently, both capable of receiving messages from the CAN bus. Normally these buffers are programmed with no filters applied so that they receive all messages, with both 11 bit & 29 bit IDs. However, filters may be applied to these receive buffers to allow through only messages with an ID that matches at least one of the filters. This is sometimes useful when looking for a particular message or when analysing a heavily loaded bus.

The filtering consists of a mask for each buffer & a set of filters, two filters for buffer 1 & four filters for buffer 2. The mask identifies which bits within the ID are relevant for filtering. If the mask bit is set to a logic '1', then this bit is used to match the corresponding bit in the received message ID with each of the filters in turn. If there is a match between all the relevant bits of the message ID & all the relevant ID bits of at least one of the filters, then the message is accepted by the receive buffer. If there is no match between the filters & the message ID then the message is rejected.



Bits within the mask that are set to a logic '0' are ignored for filtering purposes. Hence, if the mask is all zeroes, '000' (hexadecimal) for an 11 bit mask or '00000000' (hexadecimal) for a 29 bit mask, then the receive buffer accepts all messages.

For example, if the mask & filters of receive buffer 1 are configured for 11 bit ID messages & loaded with the values shown below (all values are in binary),

111**00000000** – MASK
001**00000000** – FILTER 1
010**00000000** – FILTER 2

then only messages with an ID in the range 100 to 2FF (hexadecimal) are accepted by this buffer.

Appendix C – Version Revision Notes

The notes below describe the changes necessary to upgrade an application to use the latest version of the CANdo SDK.

Version 1.0 to version 1.1	
1	No changes necessary.

Version 1.1 to version 2.0	
1	CANdoOpen(…) function now returns CANDO_USB_DRIVER_ERROR, if the USB driver is not found or is an older, incompatible version. See SDK examples for usage. CANDO_USB_DRIVER_ERROR function return code added to 'Function Return Code' table in Appendix A. Application programs that use the SDK must be re-compiled after copying across the latest version of the CANdoImport header file from the SDK. Copy the header file from the SDK project that corresponds to the application programming language, eg. CANdoImport.cs from the C# example for a Visual C# project.
2	CANdoGetVersion(…) function is available to determine if the correct libraries & USB driver are installed on the PC. See section 3 for more details. If the variables, APIVersion, DLLVersion & DriverVersion passed into this function are cleared to 0 before this function is called, then any unmodified return values indicate a missing or incompatible version of the corresponding library/driver. See SDK examples for usage. This function is new in v2.0 of the API DLL.
3	The CANdoUSB.dll & CANdo.dll files are currently only available as 32 bit files. To use these files on a 64 bit version of Windows, the WOW64 sub-system must be used. Projects written in Visual C# or Visual Basic that use the .NET Framework must be compiled to use 32 bit instructions only. To set this up in Visual Studio 2005, 2008 or 2010, open the 'Build' menu & select the 'Configuration Manager…'. In the 'Active solution platform' drop down box, click on <New…>. In the 'New platform' drop down box, select x86, then click on OK to exit. Make sure that the x86 platform is selected & then re-compile the project.

Version 2.0 to version 2.1	
1	CANdoClose(…) function now returns a status value to indicate success or otherwise in closing the connection to the CANdo device. No changes are necessary to upgrade from a previous version of the SDK.
2	CANDO_INVALID_HANDLE function return code added to ‘Function Return Code’ table in Appendix A. Application programs that use the SDK must be re-compiled after copying across the latest version of the CANdoImport header file from the SDK. Copy the header file from the SDK project that corresponds to the application programming language, eg. CANdoImport.cs from the C# example for a Visual C# project.

Version 2.1 to version 2.2	
1	CANdoGetDevices(...) function is available to determine the number & type of CANdo devices connected to the PC. See section 3 for more details. A single entry in the CANdoDevices array returned by this function may be passed into the CANdoOpenDevice(...) function to allow connection to a specific CANdo device. See SDK examples for usage. This function is new in v2.2 of the API DLL.
2	CANdoOpenDevice(...) function is available to allow a connection to a specific CANdo device. Selection maybe made by hardware type, serial number or both. See section 3 for more details. This function is new in v2.2 of the API DLL.
3	CANdoDevice variable type added. Structure type to store CANdo H/W type & serial no. info. to allow unique identification of each CANdo device. This is a new variable type in v2.2 of the API DLL.
4	CANDO_CLOSED function return code renamed CANDO_CONNECTION_CLOSED. Application programs that use the SDK must be re-compiled after copying across the latest version of the CANdoImport header file from the SDK. Copy the header file from the SDK project that corresponds to the application programming language, eg. CANdoImport.cs from the C# example for a Visual C# project.
5	The CANdoTransmit(…) function may now be called using a null pointer in the place of the 'Data' array. This is primarily intended for use when transmitting a remote frame, as no data is needed. If a data frame is transmitted & a null pointer is passed in rather than a 'Data' array, then all data is transmitted as zero.

Version 2.2 to version 2.3	
1	No changes necessary.

Version 2.3 to version 3.0	
1	No changes are necessary to use the new v3.0 API DLL with existing projects based on the SDK. However, to access the new features & for full compatibility, projects should be re-compiled, after copying across the relevant CANdoImport header file & making the necessary changes in the project file(s) for all instances of the TCANdoStatus NewFlag. See Appendix A for more details. The content of the TCANdoStatus parameters is extended in v3.0 of the API DLL.
2	CANdoGetPID(...) function is available to allow custom variants of CANdo to be identified by software based on the SDK. See section 3 for more details. This function is new in v3.0 of the API DLL.
3	CANdoRequestDateStatus(...) function is available to request the date of manufacture of a CANdo device. The date is returned within the TCANdoStatus parameters after a call to the CANdoReceive(…) function. See section 3 for more details. This function is new in v3.0 of the API DLL.
4	CANdoRequestBusLoadStatus(...) function is available to request the CAN bus load calculated by a CANdoISO device. The bus load is returned within the TCANdoStatus parameters after a call to the CANdoReceive(…) function. See section 3 for more details. This function is new in v3.0 of the API DLL.
5	CANdoClearStatus(...) function is available to clear the CANdo internal system status, similar to the CANdoRequestStatus(…) function, but without the subsequent status reply. See section 3 for more details. This function is new in v3.0 of the API DLL.

Version 3.0 to version 4.0	
1	No changes are necessary to use the new v4.0 API DLL with existing projects based on the SDK. However, to access the new features & for full compatibility, projects should be re-compiled, after copying across the relevant CANdoImport header file. This version of the SDK includes both 32 & 64 bit versions of the DLLs, allowing SDK based projects to be compiled as both 32 & 64 bit applications.
2	The functions CANdoAnalogStoreClear(...), CANdoAnalogStoreWrite(...) & CANdoAnalogStoreRead(...) are available to allow configuration of the analogue input store within the CANdo AUTO device, via the SDK. Please refer to the example projects for additional help with using these functions. These functions are new in v4.0 of the API DLL.
3	The functions CANdoTransmitStoreClear(...), CANdoTransmitStoreWrite(...) & CANdoTransmitStoreRead(...) are available to allow configuration of the CAN transmit store within the CANdo AUTO device, via the SDK. Please refer to the example projects for additional help with using these functions. These functions are new in v4.0 of the API DLL.
4	Projects written in Visual C# or Visual Basic that use the .NET Framework may now be compiled in both 32 & 64 bit variants. In Visual Studio 2005, 2008, 2010 & 2012 this is achieved by selecting ‘Any CPU’ in the ‘Configuration Manager...’, accessed via the ‘Build’ menu item.

Version 4.0 to version 4.1	
1	No changes necessary.
2	Linux SDK containing libCANdo.so API library available in this version of the SDK.