1.0 Introduction

The **CARTFT CANBUS** is a low cost and easy to use CAN/USB dongle, that could be used together with Windows or Linux O/S with two types of drivers. One Virtual COM Port driver (VCP) which acts as a standard RS232 COM port (in this mode CANBUS can replace our CAN232 and gain more bandwidth on CAN with minor or none software change). The other driver is a Direct Driver (D2XX) which uses a DLL to communicate with the CANBUS. Both drivers can not co-excist on the same PC, you have to choose one of them, however it is easy do uninstall and change from e.g. VCP to D2XX driver. CANBUS uses same type of commands (standard ASCII format) such as the CAN232 V2, though some old way of communicating (V1 Poll and Poll All) has been removed in favour of new optimized transfer speeds. The CANBUS handles both the 11bit ID format as well as the 29bit ID format, RTR frames, built in FIFO queues, extended info/error information and simple power up through a few commands, no external power is needed, it is supplied from the USB port. The CANBUS is only 60mm long, 35mm wide and 15mm thick using the latest technology of small SMD parts on both sides of the board, the power behind is an Atmel AVR, the Philips SJA1000 CAN controller and the FTDI Parallel FT245BM USB chip and that makes it very flexible in the way of handling small bursts of CAN frames at a high bus speed. This document describes the 1:st version of the CANBUS which can be updated via a bootloader via USB.

1.1 Installation

Please look at www.cartft.com for latest driver packages (VCP & D2XX). On the CANBUS site there are also in depth installation instructions of the CANBUS with the drivers. Please select one of the drivers which suite your needs the most. Maybe you just want to replace the CAN232 with a CANBUS, then choose VCP drivers and you are up running quickly. D2XX drivers needs a totally diffrent approach, though is faster in transfering CAN frames.

1.2 Testing the CANBUS

This test assumes you are using the VCP drivers. Test the CANBUS by installing it to a PC's USB port and install drivers if neccessary according to instructions on previous page under section 1.1. When the CANBUS gets power the 2 LEDs (red & green) will blink rapidly 3 times after it has been installed (about 1 second after it gets connected). Then start Windows Terminal software (or your favourite terminal sofware) and set it up to any baudrate, 8 databits, no parity, 1 stop bit, also set local echo on so you can see what you type and set the check flag so that it appends a line feed when it receive and end of line. Finally, make sure you have hardware and software handshaking off. Then make sure you are connected and press >ENTER< and it will make a new line, then press V and >ENTER< and it will print/reply Vhhss, where hh is the hardware version and ss is the software version. Now you know you have full contact with the CANBUS unit and can set it up with a CAN speed and open the CAN port, send and receive frames. Note that the green LED indicates that a CAN frame is succesfully sent or received into the CAN232 unit. Note that you must at least have 2 nodes (CANBUS is counted as one node) to send/receive CAN frames and that the CAN cable network is terminated at both ends with 120 ohms over the CANL and CANH lines plus that a twisted pair CAN cable is used. The CANBUS is set to accept all frames, so no need to set filters etc. for testing. The CANBUS can also be tested with the sample programs at www.cartft.com.

1.3 CANBUS limitations

There are of course limitations of how many frames the CANBUS can send & receive. Current version (V1011) is tested with a throughput of sending 1000 standard 11bit frames with 8 data-bytes at 125kbit CAN bitrate and the VCP drivers. The ”bottle neck” is of course the RS232 VCP drivers and the microcontroller not being able to handle more frames per second. So the CANBUS is aimed for low speed CAN networks and works very well with CAN speeds at 250kbit or less but of course it is usable up to 1 Mbit (but the bus load may not be high at these speeds or e.g. the filter has to be set to accept some of the frames). The CANBUS has software CAN FIFO queues for both sending and reception. These transmit FIFO can handle 8 frames (standard or extended) while the receive FIFO can handle 32 frames (standard or extended). Furthermore the CANBUS has also USB FIFO's built in the hardware, so it can only handle one or two command at a time, meaning before sending the next command to it, you must wait for an answer from the CANBUS unit (OK which is [CR] or Error which is [BELL]).

1.4 CAN Driver Design Guide

The CANBUS doesn't come with a Higher Level CAN driver/parser. Since many commercial development tools provide an RS232 ASYNC LIB (such as Visual Basic, Deplhi etc.) it is simple to write a simple program to "talk" to the CANBUS unit via the VCP drivers. The best way is to make a thread that handles all the communication to the CANBUS and puts all messages in FIFO queues or mail boxes depending on your application. Using the D2XX driver has a similar approach, though is faster in transfer and instead of "talking" to a COM port, you have an API (Application Programmer Interface) to a DLL instead which could be used in C, C++, VB or Delphi etc.

If you are migrating from CAN232, the CANBUS do not have the old fashioned **P** and **A** commands. Instead incomming CAN frames are sent out at once. Replies back from **t** and **T** commands has also changed, so it only replies [CR] or [BELL] instead of z[CR] or Z[CR] depending on command. This to save bandwidth on the USB side of the driver. Otherwise commands are the same.

Always start each session (when your program starts) with sending 2-3 [CR] to empty any prior comand or queued character in the CANBUS (many times at power up there could be false characters in the queue or old ones that was from a previous session), then check the CAN version with **V** command (to be sure that you have communication with the unit at correct speed) then set up the CAN speed with **s** or **S** command, then open the CAN port with **O**, then the CANBUS is in operation for both sending and receiving CAN frames. Send frames with the **t** ot **T** command and wait for a response back to see it it was placed in the CAN FIFO transmission queue or the queue was full. Incomming frames from the CAN bus will be sent out at ones. Then once in a while send the **F** command to see if there are any errors (e.g. each 500-1000mS or if you get an error back from the CAN232). If you get to many errors back after sending commands to the unit, send 2-3 [CR] to empty the buffer, then issue the commands again, if this continue allert the user or application within your program that there is a communication error (e.g. a damaged CAN tranceiver or power failure etc.).

The www.cartft.com website offers many sample programs in source code. These programs are free to use or alter to suit your needs.

1.5 Version Information

The version number of CANBUS consists of 2 versions, one for the hardware and one for the software. These two version numbers are combined into one unique version string with 5 characters starting with a V, then 2 characters for hardware and finally 2 characters for software. E.g. version V1010 indicates that it is hardware version 1.0 and software version 0.0. If we update the hardware we will increase version number of the 2 first characters and if we add or change commands or correct bugs the software version number will increase. To see if your CANBUS supports the commands in this manual, check which version number you have by sending the **V** command to the CANBUS (see under commands how it works).

2.0 Availible CANBUS ASCII Commands:

Note: All commands to the CANBUS must end with [CR] (Ascii=13) and they are CASE sensitive.

Sn[CR]

Setup with standard CAN bit-rates where n is 0-8.
This command is only active if the CAN channel is closed.

S0	Setup 10Kbit
S1	Setup 20Kbit
S2	Setup 50Kbit
S3	Setup 100Kbit
S4	Setup 125Kbit
S5	Setup 250Kbit
S6	Setup 500Kbit
S7	Setup 800Kbit
S8	Setup 1Mbit

Example: `S4[CR]`
Setup CAN to 125Kbit.

Returns: CR (Ascii 13) for OK or BELL (Ascii 7) for ERROR.

sxxyy[CR]

Setup with BTR0/BTR1 CAN bit-rates where xx and yy is a hex value. This command is only active if the CAN channel is closed.

xx	BTR0 value in hex
yy	BTR1 value in hex

Example: `s031C[CR]`
Setup CAN with BTR0=0x03 & BTR1=0x1C which equals to 125Kbit.

Returns: CR (Ascii 13) for OK or BELL (Ascii 7) for ERROR.

O[CR]

Open the CAN channel.
This command is only active if the CAN channel is closed and has been set up prior with either the S or s command (i.e. initiated).

Example: `O[CR]`
Open the channel

Returns: CR (Ascii 13) for OK or BELL (Ascii 7) for ERROR.

C[CR]

Close the CAN channel.
This command is only active if the CAN channel is open.

Example: `C[CR]`
Close the channel

Returns: CR (Ascii 13) for OK or BELL (Ascii 7) for ERROR.

tiiildd...[CR]

Transmit a standard (11bit) CAN frame.
This command is only active if the CAN channel is open.

iii	Identifier in hex (000-7FF)
l	Data length (0-8)
dd	Byte value in hex (00-FF). Numbers of dd pairs must match the data length, otherwise an error occur.

Example 1: `t10021133[CR]`
Sends an 11bit CAN frame with ID=0x100, 2 bytes with the value 0x11 and 0x33.

Example 2: `t0200[CR]`
Sends an 11bit CAN frame with ID=0x20 & 0 bytes.

Returns: z[CR] for OK or BELL (Ascii 7) for ERROR.

Tiiiiiiiildd...[CR]

Transmit an extended (29bit) CAN frame.
This command is only active if the CAN channel is open.

iiiiiiii	Identifier in hex (00000000-1FFFFFFF)
l	Data length (0-8)
dd	Byte value in hex (00-FF). Numbers of dd pairs must match the data length, otherwise an error occur.

Example 1: `t00000010021133[CR]`
Sends a 29bit CAN frame with ID=0x100, 2 bytes with the value 0x11 and 0x33.

Returns: Z[CR] for OK or BELL (Ascii 7) for ERROR.

F[CR]

Read Status Flags.
This command is only active if the CAN channel is open.

Example 1: `F[CR]`
Read Status Flags.

Returns: An F with 2 bytes BCD hex value plus CR (Ascii 13) for OK. If CAN channel isn't open it returns BELL (Ascii 7). This command also clear the RED Error LED. See availible errors below. E.g. `F01[CR]`

Bit 0	CAN receive FIFO queue full
Bit 1	CAN transmit FIFO queue full
Bit 2	Error warning (EI), see SJA1000 datasheet
Bit 3	Data Overrun (DOI), see SJA1000 datasheet
Bit 4	Not used.
Bit 5	Error Passive (EPI), see SJA1000 datasheet
Bit 6	Arbitration Lost (ALI), see SJA1000 datasheet *
Bit 7	Bus Error (BEI), see SJA1000 datasheet **

* Arbitration lost doesn't generate a blinking RED light!
** Bus Error generates a constant RED light!

Mxxxxxxxx[CR] Sets Acceptance Code Register (ACn Register of SJA1000). This command is only active if the CAN channel is initiated and not opened.

xxxxxxxx Acceptance Code in hex with LSB first, AC0, AC1, AC2 & AC3. For more info, see Philips SJA1000 datasheet.

Example: `M00000000[CR]`
Set Acceptance Code to 0x00000000
This is default when power on, i.e. receive all frames.

Returns: CR (Ascii 13) for OK or BELL (Ascii 7) for ERROR.

mxxxxxxxx[CR] Sets Acceptance Mask Register (AMn Register of SJA1000). This command is only active if the CAN channel is initiated and not opened.

xxxxxxxx Acceptance Mask in hex with LSB first, AM0, AM1, AM2 & AM3. For more info, see Philips SJA1000 datasheet.

Example: `mFFFFFFFF[CR]`
Set Acceptance Mask to 0xFFFFFFFF
This is default when power on, i.e. receive all frames.

Returns: CR (Ascii 13) for OK or BELL (Ascii 7) for ERROR.

Setting Acceptance Code and Mask registers

The Acceptance Code Register and the Acceptance Mask Register works together and they can filter out 2 groups of messages. For more information on how this work, look in the SJA1000 datasheet. In 11bit ID´s it is possible to filter out a single ID this way, but in 29bit ID's it is only possible to filter out a group of ID's. The example below will set a filter to only receive all 11bit ID's from 0x300 to 0x3FF.

Commands	*Comments*
M00006000[CR]	AC0=0x00, AC1=0x00, AC2=0x60 & AC3=0x00
m00001FF0[CR]	AM0=0x00, AM1=0x00, AM2=0x1F & AM3=0xF0

The first command tells the filter 2 to match 2 bits and if they are not set (in this case it corresponds to 0x3nn, the 3). The second command tells the nn to be don't care, so it could be from 0 to FF, though not so easy to read, since they are not placed nice in a row in memory. Filter 1 s turned off (uses AM0, AM1 & half lower AM3). The last byte in the mask could also be 0xE0 instead of 0xF0, then we filter out the RTR bit as well and you wont accept RTR frames.

V[CR]

Get Version number of both CANBUS hardware and software
This command is active always.

Example: `V[CR]`
Get Version numbers

Returns: V and a 2 bytes BCD value for hardware version and a 2 byte BCD value for software version plus CR (Ascii 13) for OK. E.g. `V1013[CR]`

N[CR]

Get Serial number of the CANBUS.
This command is active always.

Example: `N[CR]`
Get Serial number

Returns: N and a 4 bytes value for serial number plus CR (Ascii 13) for OK. E.g. `NA123[CR]`
Note that the serial number can have both numerical and alfa numerical values in it. The serial number is also printed on the CANBUS for a quick reference, but could e.g. be used in a program to identify a CANBUS so the program know that it is set up in the correct way (for parameters saved in EEPROM).

Zn[CR]

Sets Time Stamp ON/OFF for received frames only.
This command is only active if the CAN channel is closed.
The value will be saved in EEPROM and remembered next time the CANBUS is powered up. This command shouldn't be used more than when you want to change this behaviour. It is set to OFF by default, to be compatible with old programs written for CANBUS. Setting it to ON, will add 4 bytes sent out from CANBUS with the A and P command or when the Auto Poll/Send feature is enabled. When using Time Stamp each message gets a time in milliseconds when it was received into the CANBUS, this can be used for real time applications for e.g. knowing time inbetween messages etc. Note however by using this feature you maybe will decrease bandwith on the CANBUS, since it adds 4 bytes to each message being sent (specially on VCP drivers).

If the Time Stamp is OFF, the incomming frames looks like this:
`t10021133[CR]` *(a standard frame with ID=0x100 & 2 bytes)*

If the Time Stamp is ON, the incomming frames looks like this:
`t100211334D67[CR]` *(a standard frame with ID=0x100 & 2 bytes)*
Note the last 4 bytes 0x4D67, which is a Time Stamp for this specific message in milliseconds (and of course in hex). The timer in the CANBUS starts at zero 0x0000 and goes up to 0xEA5F before it loop arround and get's back to 0x0000. This corresponds to exact 60,000mS (i.e. 1 minute which will be more than enough in most systems).

Example 1: `Z0[CR]`
Turn OFF the Time Stamp feature (default).

Example 2: `Z1[CR]`
Turn ON the Time Stamp feature.

Returns: CR (Ascii 13) for OK or BELL (Ascii 7) for ERROR.