Can Monitor

Robin Cornelius

Feb 2022

Contents

[Can Monitor 4](#_Toc126586232)

[Main window 4](#_Toc126586233)

[Port and bit rate control 5](#_Toc126586234)

[Filters and scrolling 5](#_Toc126586235)

[Packet type tabs 6](#_Toc126586236)

[Packet display 6](#_Toc126586237)

[File Menu 7](#_Toc126586238)

[Preferences 8](#_Toc126586239)

[File Locations 9](#_Toc126586240)

[Default Plugins 10](#_Toc126586241)

[NMT Plugin 10](#_Toc126586242)

[Device OD Editor 11](#_Toc126586243)

[Saving DCF 12](#_Toc126586244)

[Loading DCF 12](#_Toc126586245)

[Updating Values 12](#_Toc126586246)

[PDO Injector 14](#_Toc126586247)

[Emergency Simulator 15](#_Toc126586248)

[Plugin Interface 16](#_Toc126586249)

[registerPDOS() 19](#_Toc126586250)

[decodeSDO() 19](#_Toc126586251)

[Menus 20](#_Toc126586252)

[Sending packets 20](#_Toc126586253)

[PDO 20](#_Toc126586254)

[SDO 20](#_Toc126586255)

[CanOpen 23](#_Toc126586256)

[Standard functions 23](#_Toc126586257)

[NMT 24](#_Toc126586258)

[Sync 24](#_Toc126586259)

[Emergency 24](#_Toc126586260)

[PDO – Process Data Objects 25](#_Toc126586261)

[SDO – Service Data Objects 25](#_Toc126586262)

[Guard 25](#_Toc126586263)

[CanOpen Packets 25](#_Toc126586264)

[CanOpen packets on the wire 26](#_Toc126586265)

[Emergency Message Format 27](#_Toc126586266)

[Standard error codes according to CiA DS-301 and DS-401 27](#_Toc126586267)

[Standard error register values 29](#_Toc126586268)

[Guard Protocol 31](#_Toc126586269)

[NMT Protocol 33](#_Toc126586270)

[PDO Protocol 35](#_Toc126586271)

[SDO Protocol 39](#_Toc126586272)

[Expedited Read Dictionary Object 40](#_Toc126586273)

[Expedited Write Dictionary Object 41](#_Toc126586274)

[Read Dictionary Object (segmented) 43](#_Toc126586275)

[Write Dictionary Object (segmented) 45](#_Toc126586276)

[Error/Abort codes 47](#_Toc126586277)

# Can Monitor

CanMonitor is a extendable Open Source application written by Robin Cornelius and licenced under the GPLv3.

It is designed to monitor and inject packets into a Can Open bus and using the plugin system can be extended to form full user facing applications that interact with the CanOpen bus. It is designed to make reading and understanding CanOpen packets much easier than dealing with the raw data described in the previous chapters.

## Main window

Graphical user interface, application, table

Description automatically generated

The main window shows a number of controls along the top bar and a list view shows a live view of all Can Packets on the CanOpen bus.

The Menu bar allows access to a number of features but also the plugins can add extra items to the menus when they load

## Port and bit rate control

The first items across the control bar are the setting for the Port selection of the interface to the Can Bus and the BitRate settings of the canbus

Graphical user interface, text, application

Description automatically generated

The Dropdown box next to Port will show any detected USB ports, any detected FDTI usb/serial converters using the FTDI protocol and any detected USB serial ports using known USB VID and PIDs. There is also a null://null1 entry which is a virtual Can Port for testing plugins.

If you plug in a USB device after CanMonitor is started you currently need to press the R button to refresh the list.

The Bitrate needs to match the requirements for the bus/system you are connecting to and this must be the same as the system or you will see no data packets.

The Open/Close button is used to open/close the connection to the CanDevice. The words of the button change to reflect the current state of the device. If the device is closed the button says open and pressing it will open the port.

## Filters and scrolling

Graphical user interface

Description automatically generated with medium confidence

The autoscroll check box will ensure the latest packet is always shown on the screen when any new packets arrive from the Can Bus.

The 6 other check boxes are filters for types of can packets. A tick next to the box allow that kind of packet and if it is unchecked that packet will be hidden if it arrives.

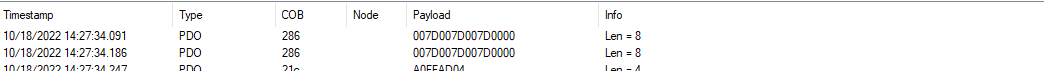
The “Clear all” button will erase all packets from the packet window

## Packet type tabs



The tab selector at the top of the packet window allows you to see all packets with “Monitor” selected, just Network Management (NMT) packets or just Emergency packets. The 4th tab is a general info window that shows debug and diagnostic information.

## Packet display



In the packet window the following formation is shown, the Timestamp when the packet was received, note due to buffering this will only be accurate to a second or so and a few packets may all appear at the same timestamp.

Type will show the type of Can Packet received :-

PDO, SDO, NMT, NMTEC, EMCY

Payload contains the contents of the packet.

Info will show the packet length in bytes unless a plugin dissector is loaded, if a dissector is loaded it will provide more information on the specific packet as many can packets are specific to the bus they are on, this is especially true of PDO packets. SDO packet handshakes are also shown in the info and again if a dissector plugin is loaded extended information can be shown.

A picture containing table

Description automatically generated

Example with a PDO dissector loaded, packets addressed to 0x485, 0x216 and 0x21c are understood and decoded. In order to understand the packet contents you need access to the EDS files for the devices on the Can Bus

Table

Description automatically generated

Example of SDO transfer, SDO transfers are a handshake so the message and the reply can be seen. Various types of SDO are supported including segmented and block transfers and all automatically decoded.

## File Menu

Graphical user interface, application

Description automatically generated

* Recent Plugins will show any recently loaded plugins (via the File menu) and allow you to quickly load them
* Load plugin will allow you to load a plugin
* Save Data will export the current packet window in xml format
* Load Data will load a previously exported xml file
* Preferences opens the preferences window
* Quit closes Can Monitor

## Preferences

The preferences entry can be accessed from the File Menu under File->Preferences

Graphical user interface, application

Description automatically generated

The plugin window allows plugins to be selected to be automatically started with software start.

* Auto Start – this will cause the program to open the port to the Can Device when the software starts at the currently selected bitrate. NB if you use a COM port then it will always try to use the same port number. If you use the USB VID/PID it will enumerate the USB devices to find the connected device.
* Limit Lines will cause the packet window to become a limited size FIFO buffer with packet no greater than the number specified being deleted when a new packet arrives.
* File log will save a log file in the folder under Desktop folder, this will save all can packets to this file.
* Close button will close the window
* Save button will save and close the window.

## File Locations

Can Monitor can run as a standalone application with out installing, or if it is installed it will live in C:\Program Files (x86)\CanMonitor (it is currently only a x86 application)

In the event of running from a standalone folder the following will be true:-

The plugins will be in a plugins subfolder

The can drivers will be in a drivers subfolder

Settings.xml will be the default settings as shown in the preferences window

Autoload.txt will be a list of plugins to load automatically at start up

If the application is installed the base folder will be :-

C:\Program Files (x86)\CanMonitor

There will also be a plugins and drivers subfolder here.

The autoload.txt file will be system wide and the application will always load plugins listed in this file.

A 2nd autoload.txt will be read from %appdata%\CanMonitor

%appdata%\CanMonitor will also contain the Most recently used plugins list and the user preferences settings.xml file

# Default Plugins

A number of default plugins are supplied with CanMonitor to perform basic and useful CanOpen functions, such as Bus stop/start/restart. Querying remote devices via SDO using their electronic data sheets etc.

## NMT Plugin

Graphical user interface, application, table, Word

Description automatically generated

The NMT plugin is a default plugin supplied with and auto loaded by CanMonitor. It adds a NMT menu to the application and has a number of menu options

The first options all issue the corresponding NMT command to the entire buss, so Start Bus will send an NMT message of Start to all nodes on the bus.

The advanced window has the same commands but also has a Node number box, node 0 will broadcast to the entire bus, any other number will only send the command to that node.

Graphical user interface, text, application, Word

Description automatically generated

## Device OD Editor

The device od editor allows the inspection and update of devices on the CanOpen bus. It uses the SDO protocol to talk to the device and can access any part of the object dictionary that is exposed via the electronic data sheet.

To start, first go to “file->Load DataSheet/Device File XDD EDF XCF” then select the device file for the specific device.

There are 2 types of file and 2 formats. The Modern format is based on XML, the old format is based on text files. The two types are the Electronic Data Sheet and the Device Configuration File. They are both very similar except the device configuration file also contains information on the current state of the device and is used to record the setup of field deployed devices, where as the electronic data sheet only records the default settings.

Graphical user interface

Description automatically generated

When a data sheet is loaded the table will be populated. The Index and Sub index of the object are shown along with the Name as defined in the datasheet, the CanOpen variable type and the default value.

The next columns are the current value (if read back from the device) and the DCF value which is the value saved in the DCF file.

If you select the Correct Node for the device in Node, then press Read the current column will populate. If you make a mistake and select the wrong device it may take a long time to complete as it will have to timeout for each non existent entry. In this case hit “Flush SDO”

Some extra options are provided :- Type, this allows a storage type to be selected. This is an extension to the standard by CanOpenNode so for a standard device that does not support the extension select “All”. If the device does support the extension the RAM,ROM and EEPROM options allow you to see different parts of the device, EEPROM is the most useful here as it references settings that will be saved at power off. Some devices require an explicit save and the “Send save req via SDO 0x1010” is for this purpose.

The Manufacture Specific only check box will hide the standard CanOpen entries from the table and only show the Manufacture specific ones.

Add Selection to custom will allow you to select a subset of the data if you are interested in watching specific network variables only. And the Auto refresh and timer will cause all the variables to be reloaded on the selected interval.

Highlighting is used to show differences between Default and Current and DCF and Current states to indicate to the user something has been changed

## Saving DCF

After pressing “Read” and the Current column is populated, you may save the current state by going to “File->Save Difference” this will save the current table as a DCF file. With all current settings recorded. This is ideal for field deployments and for saving the actual state differences compared to the default.

## Loading DCF

Loading the saved DCF file is the same as loading any other datasheet except the current values will be placed in the DCF column. In addition the Write DCF will apply all differences to the device to update it to the recorded state.

## Updating Values

To update any value (assuming it is write enabled) double click on the value

Graphical user interface, text, application, chat or text message

Description automatically generated

If the EDS file contains information about the entry it is displayed in the Box.

To update the value type in the new value in “current” and press either Update to just apply the value, or “update and close” to apply the value and close this window.

Values may be entered in Decimal format eg 10, 10.123 (depending if it is an integer or a float) or Hexadecimal format with a leading 0x eg 0x1234 0xAB12. Strings should be entered as plain text, no quotes

## PDO Injector

The PDO injector allows you to simulate PDOs on the CanOpen network. Prior knowledge of the devices and packets is required. You need to know the COB id of the PDO and how many bytes it should contain, then the individual bytes are set in the boxes B0-B7 (as enabled by number of bytes) when you are ready press Send.

Graphical user interface, text, application

Description automatically generated

Warning, sending PDOs to the network can have unpredictable effects, this is a debugging tool and packets should only be injected to networks where you have full understanding of the network and what the packets purpose is.

## Emergency Simulator

The Emergency simulator is used for injecting EMCY packets into the CanOpen bus. The sending node can be selected as well as the specific Error bits and Error codes from the predefined CanOpen standard. It is also possible to set the additional bytes for custom codes

Graphical user interface, application

Description automatically generated

Caution should be applied when using this on a real CanOpen bus it is a development tool intended for developers to test the handling of EMCY packets on their network.

# Plugin Interface

CanMonitor can be extended with plugins, as described a number are included with the system. The user can add more plugins to the system by compiling code as a .net dll linked with the appropriate interface dlls provided.

The Required interfaces are PDOInterface and libCanopenSimple, both of these are supplied with the CanMonitor application

Below is an overview of the implementation, some of the details are specific to libcanopensimple and the source code/documentation for that should be consulted in addition to this document. https://github.com/robincornelius/libCanOpenSimple

The interface looks like this :-

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using libCanopenSimple;

namespace PDOInterface

{

public interface IPDOParser

{

void deregisterplugin();

void registerPDOS();

string decodesdo(int node, int index, int sub, byte[] payload);

void setlco(libCanopenSimple.libCanopenSimple lco);

}

public interface IInterfaceService

{

IVerb[] GetVerbs(string category);

void setlco(libCanopenSimple.libCanopenSimple lco);

void preregisterPDOS(Dictionary<UInt16, Func<byte[], string>> dic);

void deregisterplugin();

void DriverStateChange(libCanopenSimple.ConnectionChangedEventArgs e);

}

public interface IVerb

{

string Category { get; }

string Name { get; }

void Action(object sender, System.EventArgs e);

}

public class InterfaceService: IInterfaceService

{

public libCanopenSimple.libCanopenSimple \_lco;

Dictionary<UInt16, Func<byte[], string>> \_dic;

Dictionary<UInt16, Func<byte[], string>> \_dic2;

public void setlco(libCanopenSimple.libCanopenSimple lco)

{

this.\_lco = lco;

}

public void deregisterplugin()

{

foreach(KeyValuePair<UInt16, Func<byte[], string>> kvp in \_dic2)

{

\_dic.Remove(kvp.Key);

}

}

public void preregisterPDOS(Dictionary<UInt16, Func<byte[], string>> dic)

{

\_dic = dic;

\_dic2 = new Dictionary<ushort, Func<byte[], string>>();

}

public void addpdohook(UInt16 cob, Func<byte[], string> functor)

{

\_dic.Add(cob, functor);

\_dic2.Add(cob, functor);

}

Dictionary<string, List<IVerb>> verbs = new Dictionary<string, List<IVerb>>();

public IVerb[] GetVerbs(string category)

{

if (verbs.ContainsKey(category))

return verbs[category].ToArray();

else

return null;

}

protected void addverb(string name, string category, Action<object, System.EventArgs> action)

{

verb v = new verb(name, category, action);

if (!verbs.ContainsKey(category))

verbs.Add(category, new List<IVerb>());

verbs[category].Add(v);

}

public virtual void DriverStateChange(ConnectionChangedEventArgs e)

{

}

}

public class verb : IVerb

{

private string \_category;

private string \_name;

Action<object, System.EventArgs> \_action;

public verb(string name,string category, Action<object, System.EventArgs> action)

{

\_action = action;

\_category = category;

\_name = name;

}

public string Category

{

get { return this.\_category; }

}

public string Name

{

get { return this.\_name; }

}

public void Action(object sender, System.EventArgs e)

{

if(\_action!=null)

\_action(sender,e);

}

}

}

For your own class the minimum requirements are a class that implements interfaces in IPDOParser, it may be desirable to derive your class from InterfaceService to save some boilerplate code.

If using InterfaceService this is the minimum code required to compile :-

public class PluginExample : InterfaceService, IPDOParser

{

public PluginExample()

{

}

public string decodesdo(int node, int index, int sub, byte[] payload)

{

return "";

}

public void registerPDOS()

{

}

}

## registerPDOS()

registerPDOS() is a function where the PDO hooks can be registered, these are callbacks that fire everytime a matching PDO is received.

The hooks use the addpdohook() function which take 2 paramaters, the COB id to listen for and the function to execute :-

addpdohook(0x182, PDO182);

addpdohook(0x181, PDO181);

addpdohook(0x183, PDO183);

the hook functions receive a byte[] parameter and should return a string of formatted data which will be shown in the CanMonitor Info column :-

public static string PDO182(byte[] data)

{

Int16 data = BitConverter.ToInt16(data, 2);

string msg = string.Format("data = {0}", data);

return msg;

}

Return null to suppress the PDO from showing in the main CanMonitor window.

## decodeSDO()

decode SDO is called when an entire SDO packet handshake is completed, this is again used to update the info column on CanMonitor but can be used for other purposes such as logging etc.

the prototype for the function is as follows :-

public string decodesdo(int node,int index,int sub ,byte[] payload)

int node, contains the SDO COB id and will be a value from 0x580 to 0x600 so the user should subtract 0x580 to get the actual node id.

Int index, this is the index of the Object Dictionary read or written to.

Int sub this is the sub index of the Object Dictionary read or written to.

Byte[] payload, this contains the data what was sent or received via the SDO handshake.

The function should return a string with any messages to add to the Info column on the main canmonitor. Or an empty string “”;

## Menus

The command adverb() allows the plugin to add menu items

protected void addverb(string name, string category, Action<object, System.EventArgs> action)

Name is the name to appear on the menu,

Category is the menu parent to add the entry to or “\_root\_” to add to the top level menu.

Action is the call back handler to receive notification the user clicked on the menu

## Sending packets

The variable \_lco provided by the InterfaceService helper class contains the reference to the libcanopensimple interface, this allows the user to send data and action SDOs eg

### PDO

public void writePDO(UInt16 cob, byte[] payload)

example :--

byte[] data = new byte[2];

data[0] = 0x20 | 0x10 | 0x02;

\_lco.writePDO(0x185, data);

### SDO

public SDO SDOwrite(byte node, UInt16 index, byte subindex, <TYPE> udata, Action<SDO> completedcallback)

public SDO Scored(byte node, UInt16 index, byte subindex, Action<SDO> completedcallback)

Note versions of SDOwrite and SDOread are provided for major datatypes

UInt32  
Int64  
UInt64  
Int32  
Int16  
UInt16  
float  
byte  
sbyte  
byte[]

Example

Byte data;

Byte data = 0xff;

lco.SDOwrite(node, index, sub, data, a => {

//Lambda callback remember if you update the UI to use Invoke!

});

If you don’t wish to use the lambda the prototype for the callback is :-

void SDOcallback(SDO sdo)

{

}

lco.SDOwrite(node, index, sub, data, SDOcallback);

# CanOpen

CanOpen is built on top of the standard CAN protocol and is used extensively in Automation. It is a well defined standard and controlled by the CiA group in the words of the CiA group “CANopen unburdens the developer from dealing with CAN hardware-specific details such as bit timing and acceptance filtering. It provides standardized communication objects (COB) for time-critical processes, configuration as well as network management data.”

CanOpen splits the network up into individual nodes and each node has multiple functions, the combination of a node ID and a function produce what is known as a COB ID. In theory CanOpen supports up to 127 individual nodes. Everything in CanOpen essentially revolves around the object dictionary, every node has its own object dictionary and this contains all the data that will be transmitted between nodes, it also contains all the configuration and settings parameters of each node. There are a certain number of standard entries that all nodes are expected to have (as defined in the standard) but then there are many optional nodes that are only present if a device implements a certain function, eg a 8 output relay module has a specific entry in dictionary for setting the output states of each channel.

CanOpen much like Can is a broadcast protocol. A particular node sends a command, the command is received by all nodes. Acceptance filters within the nodes determine if the node cares about a packet or not. Most functions combine a function ID with a node ID to produce the COB ID, the standard functions are discussed next.

## Standard functions

The standard functions in CanOpen are as follows:-

|  |  |
| --- | --- |
| **ID** | **Description** |
| 0x000 | NMT |
| 0x080 | Sync |
| 0x080 + Node ID | Emergency |
| 0x100 | Timestamp |
| 0x180 + Node ID | TX PDO 1 |
| 0x200 + Node ID | RX PDO 1 |
| 0x280 + Node ID | TX PDO 2 |
| 0x300 + Node ID | RX PDO 2 |
| 0x380 + Node ID | TX PDO 3 |
| 0x400 + Node ID | RX PDO 3 |
| 0x480 + Node ID | TX PDO 4 |
| 0x500 + Node ID | RX PDO 4 |
| 0x580 + Node ID | TX SDO |
| 0x600 + Node ID | RX SDO |
| 0x700 + Node ID | Guard |
| 0x7E4 | LSS TX |
| 0x7E4 | LSS RX |

The various functions listed above are described below in summary. Full details are provided on separate pages.

### [NMT](http://www.byteme.org.uk/nmt-protocol-network-managment-canopen/)

Network management, This packet is sent only my the master node and is used to start, stop and reset nodes on the network. Nodes have various operation states and in each state they are allowed to do certain operations.

### [Sync](http://www.byteme.org.uk/sync-protocol-canopen/)

This packet is used in combination with synchronous PDOs, the master sends the sync packet and all the slaves will respond by sending in their PDO data (that is configured as synchronous)

### [Emergency](http://www.byteme.org.uk/emergency-messages-canopen/)

This packet sends network and node critical events to all other nodes.

### [PDO – Process Data Objects](http://www.byteme.org.uk/pdo-process-data-objects-canopen/)

This protocol is used to rapidly send data from one node to 1 or more others. PDOs are very flexible and transmit particular values from one nodes object dictionary to another. The values they transmit, the update rates and behaviour are fully programmable and are themselves defined in the object dictionary,

### [SDO – Service Data Objects](http://www.byteme.org.uk/sdo-service-data-objects-canopen/)

This protocol allows you to read or write another nodes object dictionary in a polled manner. Two forms exist, a short form for expedited reads or writes where everything required fits within the 8 bytes of the CAN frame, or extended where multiple handshakes are used to transfer all the data.

### [Guard](http://www.byteme.org.uk/guard-protocol-canopen/)

This protocol is used for various guarding/heartbeat systems to ensure nodes are alive and responding. Wake up messages (when a node first powers up) also are transmitted in this protocol.

## CanOpen Packets

All CanOpen packets have the same structure,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **COB-ID** | **RTR** | **Data length** | **data** |
| Length | 11 Bits | 1 bit | 4 bits | 0-8 bytes |
| Range | 0x000-0x7FFF | 0,1 | 0-8 | 8 Bytes |

As discussed COB ID is function + node id, it has to fit in to the 11 bit can header so values 0x000-0x7FF are allowed, lower COB-IDs have higher priority, so NMT and emergency messages are more important than PDOs and SDOs etc.  
Next 1 bit is RTR, remote transmission request (RTR) this is discouraged from being used in CanOpen

Next 4 bits are the length, this specifies the length of the rest of the packet, or how many data bytes follow 0-8

Then up to 8 bytes of data (must agree with length bits)

## CanOpen packets on the wire

As a quick note, if you are observing can open packets on the wire with a scope for example, what you see is not exactly what is described previously but is more like:-

|  |  |
| --- | --- |
| **Description** | **Length** |
| Start bit (0) | 1 |
| Can Identifier or COB-ID | 11 |
| Remote Transmission request (RTR) | 1 |
| IDE (0) | 1 |
| Reserved 0 (0) | 1 |
| Data length | 4 |
| Data | 0-64 |
| CRC | 15 |
| CRC Delimiter | 1 |
| ACK slot | 1 |
| ACK Delimiter | 1 |
| End of frame | 7 |

The Pink read bytes are the ones described in this document as those are the ones you have control over at the CanOpen level, the other frames should be considered a lower level and part of CAN.

But watch out for bit stuffing. Every 5 bits of the same polarity will generate a bit of the other polarity which needs to be factored in the above table when decoding the protocol. If you are using a higher level decoder on a scope, such as Picoscope offers this will handle this for you and just give you the required data which out you worrying about the decoding, but handy to know if you ever need to use a scope!

# Emergency Message Format

The emergency message is a high priority message with the following format

| **Can header** | **rtr** | **len** | **Byte 0** | **Byte 1** | **Byte 2** | **Byte 3** | **Byte 4** | **Byte 5** | **Byte 6** | **Byte 7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x080 + node | 0 | 8 | Error Code | | Error Register | Vendor specific data | | | | |

Do not confuse SYNC and Emergency messages as SYNC has a COB ID of 0x080 where as Emergency has a COB-ID of 0x080+ Node ID.

### Standard error codes according to CiA DS-301 and DS-401

|  |  |
| --- | --- |
| **Error code** | **Description** |
| 0x00xx | error Reset or No Error |
| 0x10xx | Generic Error |
| 0x20xx | Current |
| 0x21xx | Current device input side |
| 0x22xx | Current inside the device |
| 0x23xx | Current device output side |
| 0x30xx | Voltage |
| 0x31xx | Mains Voltage |
| 0x32xx | Voltage inside the device |
| 0x33xx | Output Voltage |
| 0x40xx | Temperature |
| 0x41xx | Ambient Temperature |
| 0x42xx | Device Temperature |
| 0x50xx | Device Hardware |
| 0x60xx | Device Software |
| 0x61xx | Internal Software |
| 0x62xx | User Software |
| 0x63xx | Data Set |
| 0x70xx | Additional Modules |
| 0x80xx | Monitoring |
| 0x81xx | Communication |
| 0x8110 | CAN Overrun (Objects lost) |
| 0x8120 | CAN in Error Passive Mode |
| 0x8130 | Life Guard Error or Heartbeat Error |
| 0x8140 | recovered from bus off |
| 0x8150 | CAN-ID collision |
| 0x82xx | Protocol Error |
| 0x8210 | PDO not processed due to length error |
| 0x8220 | PDO length exceeded |
| 0x8230 | DAM MPDO not processed destination object not available |
| 0x8240 | Unexpected SYNC data length |
| 0x8250 | RPDO timeout |
| 0x90xx | External Error |
| 0xF0xx | Additional Functions |
| 0xFFxx | Device specific |

### Standard error register values

|  |  |
| --- | --- |
| **Bits** | **Description** |
| bit 0 | generic error |
| bit 1 | current |
| bit 2 | voltage |
| bit 3 | temperature |
| bit 4 | communication error (overrun error state) |
| bit 5 | device profile specific |
| bit 6 | reserved (always 0) |
| bit 7 | manufacturer specific |

The error reporting is also stored with in following Object Dictionary entries

| **Entry** | **Array** | **Meaning** | **Size (bits)** |
| --- | --- | --- | --- |
| 0x1003 | Yes | Error code (8 Values stored) | 32 |
| 0x1001 | No | Error register | 8 |

The Entry at 0x1003 is in the following format and provides an error history for the device:-

|  |  |
| --- | --- |
| **Bits** | **Description** |
| 31-24 | Manufacturer-specific error code |
| 23-16 | Error register |
| 15-0 | Error code |

# Guard Protocol

The Guarding protocol is used for boot messages, heart beat messages and to ensure that nodes are alive and healthy or to generate errors if not.

Two principles are used for guarding and this is

1. heartbeat messages, where the nodes transmits their communication state as a proof of live
2. node-guarding, where each node is queried by the NMT master to check it is alive.

With heatbeat messages nodes can be configured as heartbeat producers or heartbeat consumers, if a consumer does not see a expected heartbeat within its expected interval it signals an error

With node guarding, nodes are expected to respond to a can remote frame with in a certain time “node life time” and likewise the nodes expect to see a request from the master and if it goes missing for a certain period they assume the master as failed and sends a “Life guarding event”. If the node guarding master doesn’t receive the current state reply within the guard time, it signals a remote error for that slave. If the slave doesn’t receive this request during its ‘life time’ (guard time \* life time factor) then the slave will switch to pre-operation state.

All nodes always transmit a heatbeat when switching into pre-operation state

The packet format is as follows:-

| **Can header** | **rtr** | **len** | **State** | |
| --- | --- | --- | --- | --- |
| 0x700 + Node Id | 0 | 0001 | 1 bit toggle | 7 bits status |

Status can be

Value

| **Value** | **Meaning** |
| --- | --- |
| 0x00 | Boot |
| 0x04 | Stopped |
| 0x05 | Operational |
| 0x7f | Pre-operational |

The extra bit 0x80 is only used in the guarding protocol where it must toggle every transmission, in heartbeat mode it is fixed as 0.

The following Object Dictionary entries control the guarding/heartbeat

| **Entry** | **Array** | **Meaning** | **Size (bits)** |
| --- | --- | --- | --- |
| 0x1017 | No | Heart beat producer time in ms | 16 |
| 0x1016 | Yes | Heartbeat Consumer | 32 |
| 0x100C | No | Guard Time | 16 |
| 0x100D | No | Lifetime factor | 8 |
| 0x100E: | No | node guarding identifier  (not required to be implemented) defaults to 0x700 + node id | 16 bit? |

Heartbeat consumer (0x1016) is an array of 32bit values that describe the heartbeat monitoring configuration. each monitored node should have an entry as a sub index of 0x1016, each sub index has the following makeup:-

| **Bits** | **Meaning** |
| --- | --- |
| 31-24 | Reserved |
| 23-16 | Monitored Node Address 0x001 to 0x127 |
| 15-0 | Monitoring time (ms) |

# NMT Protocol

The NMT Protocol is used by the master node to start/stop and reset the slave nodes of the system. Depending on configuration settings (in the object dictionary) nodes may start up in run or pre-operational state. Any nodes in pre-operation state will require being set to run by the master

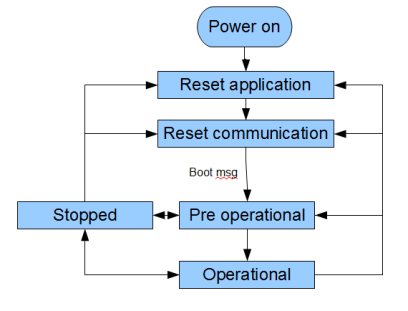
The NMT packet is aways sent with a COB-ID of 0x000 this is the highest priority message that can be sent and it has the following format.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **COB-ID** | **rtr** | **len** | **NMT Function** | **Target Node** |
| 0x000 | 0 | 2 | 1 byte | 1 byte |

Target Node can be either the specific node to address 0x01-0x7F or 0x00 to address all nodes on the bus

NMT Function can be one of the following

|  |  |
| --- | --- |
| **Code** | **NMT Function** |
| 0x01 | Enter Operational |
| 0x02 | Enter Stop |
| 0x80 | Enter Pre-operational |
| 0x81 | Reset node |
| 0x82 | Reset communication |

[](https://i0.wp.com/www.byteme.org.uk/wp-content/uploads/2015/11/canopennmtstate3.png)

After power on the node automatically enters initialization, it then moves to pre-operational and emits a Guard 0x700 Message of type bootup to signal to other nodes that it is present and awake. Depending on the nodes configuration it may then automatically enter Operation state. The Initalisation state is formed from two substates, reset application and reset communications. Reset application resets all the object dictionary entries from the standard profile and the manufacture specific sections. Reset communication resets the object dictionary entries from the communications profile to their power on values.Either of these reset states may be entered directly at any other time .

In different states various protocols are available to the node, in operational all protocols are available, in Stopped only a limited subset is available, as shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Stopped** | **Pre operational** | **operational** |
| NMT | Yes | Yes | Yes |
| EMCY | No | Yes | Yes |
| PDO | No | No | Yes |
| SDO | No | Yes | Yes |
| GUARD | Yes | Yes | Yes |
| SYNC | NO | Yes | Yes |
| TIMESTAMP | NO | Yes | Yes |

# PDO Protocol

Process Data Objects is a protocol for rapidly moving data from one node to another, data can be moved when it changes, at a preset interval or via the SYNC mechanism so multiple nodes can synchronize data gathering.

Process Data objects can be configured to move multiple object dictionary entries from one node to another. The mapping of which dictionary get end and where they end up is totally programmable. PDOs are a maximum of 8 bytes of data but in those 8 data bytes you can pick and choose which object dictionary values to send.

A simple example is you may have a device with multiple analogue inputs and if each analogue input is 16 bit you could potentially send 4 different channels at once via the PDO. Or you could send one channel and some other data that the node may have that is useful to your overall system.

The PDO (TX) protocol is very simple and is just :-

| **Can header** | **rtr** | **len** | **Byte 0** | **Byte 1** | **Byte 2** | **Byte 3** | **Byte 4** | **Byte 5** | **Byte 6** | **Byte 7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x180 + node | 0 | 1-8 | Len bytes of data | | | | | | | |

PDOs may have 1 byte of data or 8, What is contained in the data bytes and where it ends up is the devil of the detail. PDOs are configured by Object Dictionary entries and it is the understanding of these that is key to understanding and using PDOs. PDOs also are configured in pairs with a matching TX and RX PDO.

The TX PDO is controlled by two Object Dictionary entries 0x1Axx and 0x18xx. The RX PDO is controlled by two  Object Dictionary entries 0x16xx and 0x14xx. The format of the 0x1Axx and 0x16xx entries are identical and also they need to be a matched pair between the transmitting and receiving nodes

The PDO configuration registers 0x1Axx and 0x18xx simply specify which object dictionary entries will be transmitted and received with the PDO packet. It is possible to specify any OD entries that you are interested in. On the receiving side you specify where the received data will be put and again you can specify any OD entries that are of the correct size to receive the data

Up two 8 configuration bytes are available, but the total amount of data they specify must be 8 bytes or less, so you cannot for example specify 8x32bit values to transfer but you can specify 8×8 bit values

Each configuration entry has the following format and is 4 bytes long

| **Index** | **Sub** | **Length** |
| --- | --- | --- |
| 2 bytes | 1byte | 1byte |

This just specifies which OD entry to send in the packet. the packet will be populated in the order specified in the 0x1Axx entry.

The complete format of the 0x1Axx and 0x18xx entries looks like

| **Number of elements (0-8)** | **Entry 0** | **Entry 1** | **Entry 2** | **Entry 3** | **Entry 4** | **Entry 5** | **Entry 6** | **Entry 7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |

So that 0x1Axx and 0x18xx is just an array of 32 bit values.

A quick example suppose you have some sensors reading analog input, The Canopen standard will find analog input 16 bit sensors in OD entry 0x6401 and individual channels will be in sub indexes 1-8. So lets say you are interested in sending Analog channels 1 and 2 back to the host node. And lets also assume the same node also has digital inputs that you are interested in and you wish to read the 1st digital input (8 bit) this will be found at 0x6000 Sub 1. Lets also say this is the first TX PDO we wish to configure. So your TX PDO entry for 0x1Axx would look like

| **Index** | **Number of sub indexes** | **Entry 1** | **Entry2** | **Entry3** |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x1A00 | 3 | 64010110 | 64010210 | 60000108 | 0 | 0 | 0 | 0 |

Note all values in hex following the previously defined format, of index,sub,length. So in this case we have defined a PDO with 5 bytes so 3 are spare if we need them later.

You also need to configure the RX PDO mapping on the receiving node in the same way (index  0x18xx) so lets once again say its the first RX PDO we are using so we can put the data in 0x1800 we could put exactly the same as the TX PDO mapping eg :-

| **Index** | **Number of sub indexes** | **Entry 1** | **Entry2** | **Entry3** |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x1600 | 3 | 64010110 | 64010210 | 60000108 | 0 | 0 | 0 | 0 |

This would have the effect of every time the PDO was transmitted the receiving node would get an exact copy of the 6401 Sub 1 and SUb 2 and 6000 sub 1 entries in its own object dictionary. Of cause you do not need to put the data in the same object dictionaries as the TX node had them, they can go anywhere, maybe you want to keep them in some custom entries at 0x3000

| **Index** | **Number of sub indexes** | **Entry 1** | **Entry2** | **Entry3** |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x1600 | 3 | 30000110 | 30000210 | 30010108 | 0 | 0 | 0 | 0 |

Here the two analog readings are placed in 0x3000 sub 1 and 0x3000 sub 2 and the digital reading in 0x3001 sub 1. Its all down to what you need to do and wish to achieve.

Two more registers are important with the PDO mapping and these are for the tx side 0x18xx and rx side 0x14xx. These registers control the COB-ID the TX and RX nodes act on as well as a number of parameters on the TX side that effect when the PDO is transmitted.

| **Sub Index** | **Description** | **Size** |
| --- | --- | --- |
| **0** | **Number of sub indexes** | **Byte** |
| 1 | COB ID | 4 Bytes |
| 2 | Type | Byte |
| 3 | Inhibit time (ms) | 2 Bytes |
| 5 | Event timer (ms) | 2 Bytes |

For the RX PDOs only sub index 0 and 1 are used. Its important to note that the COB ID of the configured RX and TX PDO must match. So either the RX or TX PDO must be configured with a different COB ID to the default or they will never talk to each other. I usually leave the TX in its default COB ID of 0x180+node id and change the RX PDO COB ID in 0x14xx sub 1 to match.

The Type field specifies the details of when the PDO is transmitted

| **Type** | **Description** |  |
| --- | --- | --- |
| 0 | Acyclic synchronous |  |
| 1 – 240 | Cyclic synchronous |  |
| 241 – 251 | Reserved |  |
| 252 | Synchronous RTR only |  |
| 253 | Asynchronous RTR only |  |
| 254-255 | Asynchronous |  |

Types 254 and 255 are the simplest, these are transmitted when ever the value that the mapping parameter refers to changes, so in our example above if the analog input changed value a PDO would be transmitted with the new value (and also the other analog channel and the digital input info as well). The Inhibit time sub index can prevent this being updated too often and it limits the maximum update rate. The Event timer can also be used to make Asynchronous PDOs transmit automatically at a given rate when the timer expires. Giving a minimum update rate.

Cyclic Synchronous PDOs are transmitted with the SYNC message. In fact the SYNC message causes all nodes with Synchronous PDOs configured to sample their inputs and store the data, the actual data is transmitted on the next SYNC message. The SYNC message ensures that the sample time minimizes jitter between nodes as they all sample at the same time (with in limits) but there is lag to actually getting the data, so factor this in to any control loops. The Cyclic synchronous options 1-240 can be used to divide down when the PDO is transmitted so that a value of 2 is transmitted every other SYNC message etc. the device still samples on the SYNC message as before so sample lag is only ever SYNC interval behind, but the update rate can be divided down.

With Acyclic synchronous these only transmit when a Remote request from another device ‘pre-triggers’ the PDO or A device (profile) specific event ‘pre-triggers’ the PDO.

The modes involving RTR work the same way except Synchronous RTR requires a SYNC message and an RTR packet where as Asynchronous RTR only requires the RTR packet.

# SDO Protocol

The SDO or Service Data Objects provide access to the object dictionaries in each device. They are particularly useful for configuration of devices as the SDO protocol is allowed in pre-operation mode. But it is also possible to get “process” data values by polling via SDO the appropriate object dictionary entry.

SDO protocol always confirms the read/write operation.

When performing a SDO Read or Write the Index of the object dictionary entry and the sub index is always specified, the index is 2 bytes and the sub index 1 byte. if the data to be read/written is 32 bits or less then it can be done in an expedited packet thus only one transmit and one confirm receive is necessary.

SDO uses the following COB-IDs

|  |  |
| --- | --- |
| **COB-ID** | **NMT Function** |
| 0x600+node id | SDO Receive |
| 0x580+node id | SDO Transmit |

NB the TX/RX direction are from the point of view of the device. So to query a device on the network you would send a 0x600+nodeid and get back a 0x580+nodeid

The SDO Packet looks like the following :-

| **Can header** | **rtr** | **len** | **Byte 0** | **Byte 1** | **Byte 2** | **Byte 3** | **Byte 4** | **Byte 5** | **Byte 6** | **Byte 7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x600 + node | 0 | 8 | Command | Index | | Sub Index | Data | | | |

The Can header consists of the COB ID (Function + Node), the RTR bit and 4 bits representing the packet length (0-8) This makes the header 16 bits. So the total packet is 10 bytes.

The SDO packet always contains 8 bytes of data (even if they are not all used). Command specifies the nature of the transfer read/write etc. The Index of the object dictionary being queried is the next 2 bytes (Don’t forget its Little Endian on the wire), followed by 1 byte specifying the sub-index. The remaining 4 bytes contain the data of the transfer (or zero if they are not required)

The command byte bits all have various meanings and the 8 bits of the command can be divided into bits with the following meanings depending on who sent the message. In this context the server is the node initiating the read/write operation and the client is the responding node.

If there is 4 bytes or less than the transfer can be expedited and all the data sent within the command or response packet. This limits the overall SDO transfer to just two packets. If more than 4 bytes of data is required to be transferred then a segmented transfer is used. Where after the first command or response packet, additional packets are sent and requested/acknowledged until all data is complete. There is also a Block transfer where instead of confirming each segment, the entire block is transferred then only one confirmation is made at the end.

As the various flags and handshakes are subtly different, details for each is provided.

### Expedited Read Dictionary Object

Request Server -> Client  (0x600 + Node ID)

| **Can Header** | **rtr** | **len** | **Byte 0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x600 + node | 0 | 8 | Command | Index | | Sub Index | 0 | | | |

|  |  |  |
| --- | --- | --- |
| **Command Code Bits** | **Value** | **Meaning** |
| 7-5 | 010 | CCS – Client Command Specifier |
| 4-0 | 00000 | Not used |

The overall command for this transfer is

|  |  |
| --- | --- |
| **Command Code** | **Meaning** |
| 0x40 | Read Dictionary Object |

Note! at this point you do not actually know it is an expedited transfer you have just requested to read a dictionary object, the server will confirm in the response if it can expedite the transfer. If the total data size is 4 bytes or less, the server will set the appropriate bits in the response and send the data with the response. if you have requested an object that is larger that 4 bytes the reply will have the expedited transfer bit set to 0 and then you must look at the segmented transfer

Response, Client -> Server (0x580 + Node ID)

| **Can Header** | **rtr** | **lrn** | **B0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x580 + node | 0 | 8 | Command | Index | | Sub Index | Data | | | |

The index and Sub index are as you specified and the command byte has the following meaning:-

|  |  |  |
| --- | --- | --- |
| **Command Code Bits** | **Value** | **Meaning** |
| 7-5 | 010b | SCS – Server Command Specifier |
| 4 | 0 | (Not Used) Segment toggle bit |
| 3-2 | (n) | Data size  n=3 (11b) 1 data bytes sent  n=2 (10b) 2 data bytes sent  n=1 (01b) 3 data byte sent  n=0 (00b) 4 data byte sent |
| 1 | 1 | expedited transfer |
| 0 | 1 | data set size is indicated |

|  |  |
| --- | --- |
| **Command Code** | **Meaning** |
| 0x43 | Read Dictionary Object reply, expedited, 4 bytes sent |
| 0x47 | Read Dictionary Object reply, expedited, 3 bytes sent |
| 0x4B | Read Dictionary Object reply, expedited, 2 bytes sent |
| 0x4F | Read Dictionary Object reply, expedited, 1 bytes sent |

### Expedited Write Dictionary Object

Request Server -> Client  (0x600 + Node ID)

| **Can Header** | **rtr** | **len** | **B0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x600 + node | 0 | 8 | Command | Index | | Sub Index | Data | | | |

|  |  |  |
| --- | --- | --- |
| **Command Code Bits** | **Value** | **Meaning** |
| 7-5 | 001b | SCS – Server Command Specifier |
| 4 | 0 | (Not Used) Segment toggle bit |
| 3-2 | (n) | Data size  n=3 (11b) 1 data bytes sent  n=2 (10b) 2 data bytes sent  n=1 (01b) 3 data byte sent  n=0 (00b) 4 data byte sent |
| 1 | 1 | expedited transfer |
| 0 | 1 | data set size is indicated |

The overall command for this transfer is one of :-

|  |  |
| --- | --- |
| **Command Code** | **Meaning** |
| 0x23 | Write Dictionary Object reply, expedited, 4 bytes sent |
| 0x27 | Write Dictionary Object reply, expedited, 3 bytes sent |
| 0x2B | Write Dictionary Object reply, expedited, 2 bytes sent |
| 0x2F | Write Dictionary Object reply, expedited, 1 bytes sent |

it is important that the the above data size is correct and the correct command code is used depending on the size of the object dictionary entry you are trying to write to. Can Open will enforce that only a 1 byte write can be performed to a 1 byte entry in the dictionary and the same goes for 1,2,3 and 4 bytes so make sure you use the correct size or an error will be returned

Response, Client -> Server (0x580 + Node ID)

The response packet again contains the index and sub index you specified and should look like:-

| **Can Header** | **rtr** | **len** | **B0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x580 + node | 0 | 8 | Command | Index | | Sub Index | 00000000 | | | |

|  |  |  |
| --- | --- | --- |
| **Command Code Bits** | **Value** | **Meaning** |
| 7-5 | 011 | SCS – Server Command Specifier |
| 4-0 | 00000 | Not used |

### Read Dictionary Object (segmented)

The read starts exactly the same way as documented in Expedited Read Dictionary Object :-

Request Server -> Client  (0x600 + Node ID)

| **Can Header** | **rtr** | **len** | **B0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x600 + node | 0 | 8 | Command | Index | | Sub Index | 00000000 | | | |

The difference is in the response

Response, Client -> Server (0x580 + Node ID)

| **Can Header** | **rtr** | **len** | **B0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x580 + node | 0 | 8 | Command | Index | | Sub Index | Len | | | |

|  |  |  |
| --- | --- | --- |
| **Command Code Bits** | **Value** | **Meaning** |
| 7-5 | 010b | SCS – Server Command Specifier |
| 4 | 0 | Not Used |
| 3-2 | 00 | Not Used |
| 1 | 0 | expedited transfer |
| 0 | 1 | data set size is indicated |

Instead of any data being returned with the response, the data bytes contain a 32bit length that specifies the total data size that needs to be transferred.

Then each data “segment” is requested and returned one at a time

Request Server -> Client  (0x600 + Node ID)

| **Can Header** | **rtr** | **len** | **B0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x600 + node | 0 | 8 | Command | Index | | Sub Index | 00000000 | | | |

|  |  |  |
| --- | --- | --- |
| **Command Code Bits** | **Value** | **Meaning** |
| 7-5 | 011b | CCS – Client Command Specifier |
| 4 | 0/1 | Toggle bit, must be flipped each request (start with a 0) |
| 3-2 | 00 | Not Used |
| 1 | 0 | Not Used |
| 0 | 1 | Not Used |

Response, Client -> Server (0x580 + Node ID)

| **Can Header** | **rtr** | **len** | **B0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x580 + node | 0 | 8 | Command | Data | | | | | | |

|  |  |  |
| --- | --- | --- |
| **Command Code Bits** | **Value** | **Meaning** |
| 7-5 | 000b | SCS – Server Command Specifier |
| 4 | 0/1 | Toggle bit, flipped each request |
| 3-1 | 000 | (number of data bytes ([8-n to 7]) that do NOT contain data)  or zero if segment size not specified |
| 0 | 0/1 | 1 = Last segment |

### Write Dictionary Object (segmented)

If uploading more than 4 bytes to a client a segmented transfer can be used

Request Server -> Client  (0x600 + Node ID)

| **Can Header** | **rtr** | **len** | **B0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x600 + node | 0 | 8 | Command | Index | | Sub Index | Len | | | |

Instead of sending any data bytes, the total length of the data is sent in the Len Field of the first packet

|  |  |  |
| --- | --- | --- |
| **Command Code Bits** | **Value** | **Meaning** |
| 7-5 | 001b | CCS – Client Command Specifier |
| 4 | 0 | Not Used |
| 3-2 | 00 | Not Used |
| 1 | 0 | expedited transfer |
| 0 | 1 | data set size is indicated |

The client then responds with the following packet

| **Can Header** | **rtr** | **len** | **B0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x580 + node | 0 | 8 | Command | Index | | Sub Index | 00000000 | | | |

With command as follows:-

|  |  |  |
| --- | --- | --- |
| **Command Code Bits** | **Value** | **Meaning** |
| 7-5 | 011 | SCS – Server Command Specifier |
| 4-0 | 00000 | Not used |

Then the handshake of the data packets(segments) begins

| **Can Header** | **rtr** | **len** | **B0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x600 + node | 0 | 8 | Data | | | | | | | |

|  |  |  |
| --- | --- | --- |
| **Command Code Bits** | **Value** | **Meaning** |
| 7-5 | 011 | SCS – Server Command Specifier |
| 4 | 0/1 | toggle (change each packet, start with 0) |
| 3-0 | 0000 | Not used |

And the client confirms each segment with the following :-

| **Can Header** | **rtr** | **len** | **B0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x580 + node |  | 8 | Command | 00000000000000 | | | | | | |

|  |  |  |
| --- | --- | --- |
| **Command Code Bits** | **Value** | **Meaning** |
| 7-5 | 001 | SCS – Server Command Specifier |
| 4 | 0/1 | toggle, changes each time |
| 3-0 | 0000 | Not used |

### Error/Abort codes

When something goes wrong, or it is necessary to abort the command specifier in all cases can be changed to Abort Transfer. There are two abort messages one to the client and one from the client.

If the server needs to abort the transfer the packet is

| **Can Header** | | | **B0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x600 + node | 0 | 8 | Command | Index | | Sub index | Additional Info | | Error code | Error class |

If the client aborts the transfer the packet is

| **Can Header** | **rtr** | **len** | **B0** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** | **B7** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x580 + node | 0 | 8 | Command | Index | | Sub index | Additional Info | | Error code | Error class |

The command bits are common to both packets for an abort

|  |  |  |
| --- | --- | --- |
| **Command Code Bits** | **Value** | **Meaning** |
| 7-5 | 100 | SCS/CCS – Abort |
| 4-0 | 00000 | Not used |

The Fields Add Info, Err code and Err class describe the reason for the abort. some errors need an Additional Code

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Error Class** | **Error Code** | **Additional Code** |
| Toggle bit not alternated | 5 Service Error | 3 Parameter Inconsistent | 0 |
| Command specifier not valid | 5 Service Error | 4 Illegal Parameter | 0 |
| Object does not exist | 6 Access Error | 2 Object non-existent | 0 |
| Attempt to read a write only Object | 6 Access Error | 1 Object access unsupported | 0 |
| Attempt to write a read only Object | 6 Access Error | 1 Object access unsupported | 0 |
| Index value is reserved for further use (00A0h-0FFFh and A000h-FFFFh) | 6 Access Error | 4 Invalid address | 0 |
| Access failed due to hardware | 6 Access Error | 6 Hardware fault | 0 |
| Sub-index does not exist | 6 Access Error | 9 Object attribute inconsistent | 11h |
| Object length too high | 6 Access Error | 7 Type conflict | 12h |
| Object length too low | 6 Access Error | 7 Type conflict | 13h |
| Data cannot be transferred / Invalid signature | 8 Other Error | 0 | 20h |
| Parameter value out of range | 6 Access Error | 9 Object attribute inconsistent | 30h |
| Sub-parameter value out of range | 6 Access Error | 9 Object attribute inconsistent | 33h |
| Maximum value < Minimum value | 6 Access Error | 9 Object attribute inconsistent | 36h |
| Object cannot be mapped to PDO | 6 Access Error | 4 Invalid address | 41h |
| PDO length exceeded | 6 Access Error | 4 Invalid address | 42h |
| General internal incomptibility | 6 Access Error | 4 Invalid address | 44h |