Black Oak Engineering

Application Note: Design of a Device Controller using Microchip dsPIC33F & the CAN / CANopen Stack

A Guide for the Perplexed

Contents

Emperor Joseph II: "That is too fine for my ears – there are too many notes."
W. A. Mozart: "There are just as many notes as there should be, your majesty."
[Perhaps apocryphal.]

www.blackoakeng.com
Automation, controls, electronic design, instruments & sensors, software.
Brooklyn, NY, USA. Since 2003.
"It just works."

1. Introduction

This Application Note details the use of a Microchip dsPIC33F controller, which implements a CAN / CANopen communication stack. The module itself controls an array of sensitive photonic devices using a PI method. The physics and operational details for the photonic devices have been reported elsewhere. For our purposes here it suffices to know that there are eight of them in an array and that their radiance drifts considerably due to self-heating. The user requires 4 ms updates, and there are two other nodes also being updated at this rate. This is managed through periodic Sync messages. We designed a module based on a Microchip dsPIC33F 16-bit digital signal controller, the associated Microchip CAN driver chipset, and a precise high voltage current driver ensemble. The dsPIC33F is programmed in C, using the new MPLAB XC IDE.[1]

This document describes the CAN and CANopen standards generally for context, and then more specifically to justify and document design decisions for the project. It is important to note that one must learn the ground rules well before attaching a custom device to the CAN / CANopen network. There are expectations to meet. One may easily miscommunicate or trigger network fault states through inattention to detail.

The general structure of this Application Note is to start with the basics and then gradually descend 'into the weeds.' There is a glossary at the end for the numerous acronyms. This document may also serve as a useful reference for other CAN / CANopen implementations. Note, the implementation here strives to be minimal; it is the least amount of investment necessary in order to participate on the network. As always, there are tradeoffs. As a general rule, we try to keep abstraction and generality to a minimum for a given application. This is somewhat different from the Microchip Application Notes, which are quite well written but serve a somewhat different purpose. Since our primary goal is to get the customer's application running quickly and reliably, we tend to keep data structures and algorithms as simple as possible. This also minimizes bugs and unintended consequences, and it facilitates future improvement.

[1] Black Oak Engineering is a certified Design Partner for Microchip Technology®. We are fully independent of that company, and this Application Note was written independently and with complete objectivity. Any errors or omissions are entirely our own.

2. CAN basics

Controller Area Network is an efficient, robust multidrop serial communication system that lends itself well to interfacing sensors, actuators and motors. It was first developed by Bosch for such critical automotive applications as Antilock Brake Systems. It is now also widely used in medical instrumentation, aviation, robotics, military systems, and industrial automation. CAN is highly noise immune and it minimizes the cost of the physical transmission line. It does not require a Master, but effectively the host node often assumes this role, especially when using CANopen. In the present case the host is the customer control station, which is running Matlab / Simulink.

Some salient features of CAN generally:

- The basic CAN message package includes either an 11 or a 29 bit message ID and between 0 and 8 data bytes, plus a modest amount of other packaging bits. This is much leaner than, say, TCP/IP.
- Messages are all broadcast asynchronously to the network. In other words, the network is multi-master or peer-to-peer. Again, this flatness is restricted under CANopen.
- The basic signal is a differential voltage. When quiescent the two lines should present as a +2.5V level, with respect to cable ground. This is a logic low level. During a logic high pulse the 'high' line will rise to 3.5 - 4.5 V and the 'low' line will fall to 0.5 - 1.5 V, with a 3.0 to 4.0 V differential. This is a necessary condition. The network will not function in the absence of conditions close to this.
- Topologically the network is expected to be linear, with nominal 120 Ω[2] termination resistors at both ends. Short stubs off of the line are usually acceptable. If the signals on one's network are significantly different from the nominal states described here, the first thing to check is the two terminations.
- There is built-in message collision arbitration. A message with higher priority automatically, *electrically* overrides another. Thus, a binary 0 is termed *dominant,* a binary 1 *recessive,* by rough analogy with genetics. The lower the numerical value of its message ID, the higher its priority, as a zero will 'dominate' the open-drain bus configuration.
- One sign of poor transmission line quality is that, when the node makes its first message (such as a Boot-up CANopen message), the node repeats itself endlessly. The node is not necessarily malfunctioning. It may be assuming that its message did not transmit because another node with higher priority overrode it. This is a warranted assumption, although not at all desirable behavior.
- Message identifiers range from index 0 to 1023. Generally speaking, in CAN there are no Node IDs per se. Rather, each message has its own identifier, and a node[3] responds to

[2] 120 Ω is the characteristic impedance of typical twisted pair transmission line (especially as used by RS-485 networks). Some networks require slightly different values. An electromagnetic wave traveling down the transmission line will reflect off of any impedance mismatch. This manifests itself as corrupted signal.

[3] In this document I use node and device inter-changeably.

that ID as appropriate. There may thus be any number of nodes, although 64 is a practical limit. In CANopen, however, we specifically limit the number of nodes to 127, and they are assigned IDs. Node ID = 0 is reserved for the CANopen host.

- As a normal, best practice guideline, CAN networks should be kept idle 50% of the time.

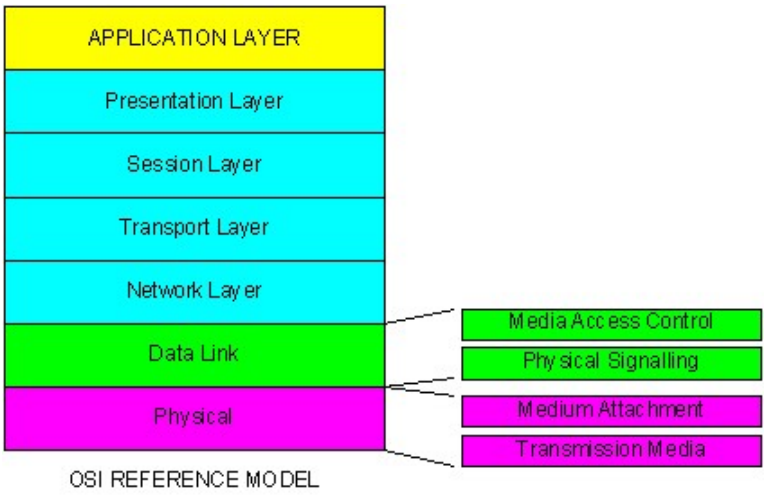

Figure 2.1 – OSI Communication Stack

Figure 2.1 – OSI communication stack

Figure 2.1 depicts the standard OSI model of a communication stack, or 'reference model'. CAN bus by itself occupies the Physical and Data Link layers, while CANopen is one way to occupy the higher layers. Other high-level protocols include DeviceNet and SAE J1929 and ARINC 825. It is noteworthy that CANopen may also be based on other physical media, including RS485 and EtherCAT.

CAN Physical and Data Link layers are described in more detail in section 8. The official CANopen standard, which is an open standard, is governed by the CAN in Automation group, CiA, under their DS-301 and associated publications. There is a large community of practitioners and hardware providers. CANopen is an open standard, and it allows for individual design decisions. CiA is an active governing oversight committee. The CAN / CANopen combination fairly well encompasses the entirety of the OSI Stack. This cannot be said of, say, the Ethernet + TCP/IP combination, which requires a large amount of aftermarket proprietary middleware to specialize it to tasks such as ours. Particularly in the presentation and application layers, CAN/CANopen is designed and ready-made for automation tasks. It is limited in bandwidth and node count, but this can often be easily remedied by running parallel CAN networks. The 1 Mb/s limit (at minimal cable length) is problematic for many users. There have been efforts from Bosch and others to develop a standard and hardware set with an increased bandwidth (e.g. FlexRay), but this has not at all succeeded to the same degree as CAN usage. If

only for automotive engineering, regard this as a matter of great interest and probable change.

CAN networks are deterministic. A node which must report a message will continue to attempt transmission until it succeeds. Only nodes with higher priority may delay this. Under CANopen there are numerous mechanisms for 'Heartbeats' and such continuous self-checking. Every node is required to monitor network conditions and report faults.

3. Object Dictionary

The Object Dictionary (OD) is an important CANopen concept. It refers to the entire public interface (effectively, the 'API') of a device at a node. We may write Device Parameters to it (such as a desired output level or velocity-loop parameters), and the node is expected to read this and respond appropriately. We may read a Device Status (such as a revision level code or temperature).

In this project we structured the Object Dictionary as a simple database. In any OD each object (datum) is identified uniquely by a 16-bit index with an 8-bit sub-index. Most objects represent simple data types, such as 16-bit integers, 32-bit floats, strings, and so on. Note, this 24-bit address space exists only in abstraction. It is not the same space as that mapped by a CAN message with its 11-bit base or 29-bit extended address. *That* index is generally referred to as the 'COB-ID'. The OD index and subindex refer to points in the abstract Object Dictionary space. At the risk of introducing even more abstraction and confusion, it is important to note that the project software database lists every datum with its own linear index.

In this project the database easily swelled to over 100 indexed entries. About two-thirds of these were parameters that need to be exposed to the user for the application. These are either read-only or read-writes. The other third were dedicated either to required CANopen communication objects or to the 'meta-structure' which attends the device parameters.

For example, say that Index 6004h has five subindices under it. The subindices contain four 'objects': temperature, pressure, humidity, pH. The 0-th[4] subindex will always be the number of objects following, in this case 4. So this section of the OD looks like:

Index	*Subindex*	*Object*	*Type*
…			
6004h	.0	4	unsigned8
	.1	Temperature	float32
	.2	Pressure	float32
	.3	Humidity	float32
	.4	pH	unsigned16
6104h…			

Every device node must maintain a subset of the total OD space which includes all of the parameters that pertain to it, and a certain amount of CANopen communications overhead. This Application subset may be quite small, for a

[4] If an OD index points to only one object, its subindex is 0, and the object is addressed as *index.0.*

solitary temperature sensor, say. A full-blown DC servo motor controller will typically expose hundreds of Application parameters, in addition to the required CANopen overhead.

The 24-bit address space of the OD seemingly affords room for 2^{24} > 16 million objects, but this is not really so. For example, only certain ranges may be used for certain purposes, such as Application parameters. Moreover, several ranges have an implicit offset that must be respected. Every node has a unique Node-ID. In the example above, the Node-ID is 04h. Thus, if the network starts placing Application parameters at 6000h (not uncommon), then we maintain our node at 6004h, with the necessary subindices. Other nodes do the same at 6001h, 6002h, and so forth, with parameters that are appropriate to them. Only the host requires scope of every node's OD.

All Object Dictionaries are structured thus:

Index	***Object types***
0000h	Reserved
0001-001F	Static data
0020-003F	Complex data
0040-005F	Manufacturer specific complex data types
0060-007F	Device profile specific static data
0080-009F	Device profile specific complex data
00A0-0FFF	Reserved
1000-1FFF	Communication profile area (per CiA DS-301)
2000-5FFF	Manufacturer specific profile area
6000-9FFF	Standardized device profile area (DS-402 governs servo drives)
A000-FFFF	Reserved

Table 3.1 **- General Object Dictionary address space**

In the Static Data section CANopen requires that we explicitly define the data types in order. Typically Index = 1 defines a Boolean, 2 an 8-bit integer, 2 a 16-bit integer, etc. Unusual data types, such as 24-bit integers, follow. While conceivably the data types and sizes could be reconfigured here, we have never seen this done.
In the Communication profile area every CANopen device is required to support the following data structure.

Index	*SubIndex*	*Type (type index from Static data area)*	*Description*
1000h	0h	Unsigned32 (7)	Device type information
1001	0	Unsigned8 (5)	Error register
1017	0	Unsigned16 (6)	Heartbeat time
1018			Identity object
	0	Unsigned8 (5)	= 4, the number of subindex entries to follow
	1	Unsigned32 (7)	Vendor ID
	2	Unsigned32 (7)	Product code
	3	Unsigned32 (7)	Revision number
	4	Unsigned32 (7)	Serial number

Table 3.2 – Required Communication Profile parameters

Every device is expected to have a formal Device Profile published in which parameters particular to it may be found. This will be a data structure similar to that in the Communication profile, although it is based typically at index 6000h. Thus the Device Profile is a data structure that may be read as part of the device's Object Dictionary. It contains any parameters of interest for the device. Our present project falls under the category of *CiA 401 V3.0.0: CANopen device profile for generic I/O modules*, and has been set up accordingly. The Device Profile is expected to be published in an Electronic Data Sheet (EDS). The format of the EDS may be in a generic Microsoft '.ini' type format, or in an XML format. CiA specifies formats for a wide variety of industrial devices.

As mentioned above, although CAN by itself is an inherently democratic, peer-to-peer network, CANopen tends to enforce a hierarchy, with the host as client and nodes as servers. Much of the prescribed CANopen format serves to streamline communications, which may otherwise be unruly and inefficient. Nonetheless, it is permissible and quite common to enable two nodes to communicate directly with each other. A typical example is to have an angular encoder communicate directly with a servo drive. Again, CANopen is quite flexible and extensible. However, liberties taken with the standard and norms, while expedient today, may cause major reliability problems as hardware and software changes are made in the future.

```
    // Device type
odDb[0].index = 0x1000;             odDb[0].subdex = 0;
odDb[0].daType = DT_UNSIGNED32;     odDb[0].msb = (unsigned long *)&copDeviceType;
odDb[0].permission = DB_READONLY;   strcpy(odDb[0].descriptor, "Device type");
```

Figure 3.1 – OD source code fragment

Figure 3.1 shows a fragment from the project source code for the Object Dictionary. The OD consists of a long series of similar entries. The database is simply an array of structures. The array index is of course distinct from the OD index (1000h in the example). We also see the OD subindex, the #defined data type, a void pointer for the actual datum in physical memory, the read-write permission, and a human readable descriptor (for local database maintenance). The void pointers are cast as necessary for a wide variety of data types.

Not implemented in this project, but often desirable, is a set of upper and lower bounds. If the CANopen network attempts to write an unacceptable value to an object (assuming it to have write permission) it is often wise to limit the possibilities. The OD is the natural place to store these limits.

Another practice that is worth consideration here is the linked list. In this common practice, every entry includes a field which points to the next entry. All entries are thus linked together in a chain. This can often be an effective algorithm, especially for lists that frequently get reordered. We elected not to do linked lists in this application, however, as scanning through the linear database index is much faster. One tool that we did find highly useful though is to write a MS Windows / Linux utility program that allows user friendly editing of the Object Dictionary, with addition, deletion, and re-indexing. The text output of this program is simply cut and pasted into the project source code. This practice also lends itself well to the requirement that nodes publish a formal XML Device Profile, as described above.

Generally speaking, an Object Dictionary is expected to support dynamic reprogramming. This is explained further below, but the idea is that the host may use Service Data Object (SDO) messages to reconfigure the node's OD overall structure, not just individual entries. This necessitates a fair amount of abstraction and check mechanisms. The project described herein supports dynamic reprogramming, although it imposes some strict constraints. We have never seen a project where the OD had to be radically reconfigured dynamically, although by no means are we suggesting that this is exceptional. Dynamic reprogramming does require, without doubt, a considerable amount of overhead to support reliably.

4. Service Data Objects

There are two CANopen protocols for interacting with the Object Dictionary, the Service Data Object (SDO) and the Process Data Object (PDO), which we treat in section 5. Again, when using the CANopen upper layers of the communication stack, we lose the democratic sense of simple CAN bus. There is a network host ('master'), and it operates in a client-server relationship with the node devices. Generally speaking, only the host coordinates SDOs and PDOs, but it is quite common to have nodes communicating directly with one another via PDOs.

Note, regardless of whether they are carried by PDOs or SDOs, data types larger than one byte are transmitted in little-endian fashion. This obtains

only at the byte level for individual data types. Strings, for example, are transmitted first character first. OD Index addresses are of type unsigned16, so they are transmitted as LSB, MSB.

Using SDOs, any data object of any size in the device OD may in principle be accessed[5]. Because of this flexibility, there is significantly more communication overhead. The communication transfer must always be confirmed, or else a system error will result. SDOs are considered *reads* when data transfers from device node to host. *Write* SDOs transfer from host to node. SDOs are typically used for device initialization and configuration. They usually bear low priority message IDs (i.e., high numerical value). A network typically needs at least one SDO in order to access a node's OD. It is possible to transmit lengthy messages via SDOs by means of 'segmented transfer.' The alternative is to use 'expedited' SDOs, which contain 8 data byes, whether or not all bytes are utilized. This is elaborated below.

It is inherently inefficient to use SDOs. Say we want to transfer an angular position reading from an encoder (node *1h*) to motor controller (node *5h*). The host sends out an SDO addressed to node *1h*; *1h* sends back its SDO confirming response; the host sends out an SDO addressed to node 5h; *5h* responds. Even assuming expedited, non-segmented message content and no bus errors, this will consume hundreds of bus cycles.

Also, such a system is based on polling from the host. In many situations, the bus traffic will be endlessly repeating the same information. It is generally better to have a device programmed to only transmit when one of its parameters has changed. We may do this by setting up a PDO.

CiA uses curious, somewhat ambiguous terminology regarding inputs and outputs. A CANopen device always has two roles:

- The device performs some function in the real world. An encoder or a thermometer, for example, is an input sensor for the control system at large. A motor or actuator is an output to the control system.
- It is a node in a communication network, from which it receives input and to which it outputs. From the point of view of the network, inputs transmogrify into outputs at some point simply as part of the mechanics of CANopen.

For the most part, CiA uses 'input' and 'output' in the first sense, as in a typical control system.
There are four types of SDO supported:

1. An expedited read of 4 or fewer bytes
2. An expedited write of 4 or fewer bytes
3. A segmented read of >4 bytes
4. A segmented write of >4 bytes

There is an important point to make regarding COB-IDs. CANopen requires that all messages be either Broadcast messages (usually from the host), which go out to everyone, or that they map uniquely to one and only one node. In the latter case the Node-ID is encoded as an offset in the COB-ID.

1. An expedited read SDO is initiated by the host transmitting this message. This is step 1.

[5] In practice, some objects may not be accessible due to some system constraint. For example, a motor driver may not allow certain parameters to be altered while motion is occurring.

COB-ID	600h + Node-ID
Byte 1	40h
Byte 2	Object Index, LSB
Byte 3	Object Index, MSB
Byte 4	Subindex
Byte 5	00h
Byte 6	00h
Byte 7	00h
Byte 8	00h

The device at Node-ID responds with this message. This is step 2.

COB-ID	580h + Node-ID
Byte 1	42h, 4Fh, 4Bh, 43h
Byte 2	Object Index, LSB
Byte 3	Object Index, MSB
Byte 4	Subindex
Byte 5	Data, LSB first
Byte 6	
Byte 7	
Byte 8	

The contents of Byte 1 are detailed in Table 3.1. An important Object Dictionary datum here resides at location 2111h, the Size Indicator (SZI), which holds the number of bytes to transfer. This parameter may optionally be used to control transfers. There is a protocol for using SZI, and one for not using it. CANopen, again, is rather flexible. Indeed, it is inherently somewhat ambiguous how to detect some transfer faults. This is not a problem as long as the entire network follows the same protocols, but it requires vigilance to assure this.

2. A segmented read SDO is initiated by the host transmitting this request. This is step 1.

COB-ID	600h + Node-ID
Byte 1	40h
Byte 2	Object Index, LSB
Byte 3	Object Index, MSB
Byte 4	Subindex
Byte 5	00h
Byte 6	00h
Byte 7	00h
Byte 8	00h

This is the same as an expedited read SDO. Why? Because we may not know *a priori* how much data we are asking for. Only the node knows for sure.

The device at Node-ID responds with this message. This is step 2.

COB-ID	580h + Node-ID
Byte 1	40h, 41h
Byte 2	Object Index, LSB
Byte 3	Object Index, MSB
Byte 4	Subindex
Byte 5	00h, or number of bytes to transfer, depending upon the mode used.
Byte 6	
Byte 7	
Byte 8	

Then the host confirms that it is ready for data. This is still step 2.

COB-ID	600h + Node-ID
Byte 1	60h
Byte 2	00h, all 7 bytes
Byte 3	
Byte 4	
Byte 5	
Byte 6	
Byte 7	
Byte 8	

Then the device replies with:

COB-ID	580h + Node-ID
Byte 1	See Table 4.1
Byte 2	Data, LSB first
Byte 3	
Byte 4	

Byte 5	
Byte 6	
Byte 7	
Byte 8	

The read SDO Byte 1 commands are:

Step	***Byte 1***	***Meaning***
1	40h	Used by host to initiate a read SDO. Does not include a size indication. Used by a node when replying to host's initiate command when SZI = 0 and there are more than 4 bytes to transfer.
	41	Used by a node when replying to a read initiate and there are more than 4 bytes to transfer. Bytes 5-8 will indicate the number of bytes the node has to transfer. Only valid if SZI ≠ 0, otherwise node uses 40h.
	42	Used by a node when replying to a read command with ≤4 bytes. Actual number of bytes is not indicated. Only valid if SZI = 0.
	4F	Used by a node when replying with exactly 1 data byte. We use only Byte 5. Only valid if SZI ≠ 0, otherwise node uses 42h.
	4B	Used by a node when replying with exactly 2 data bytes. We use only Bytes 5 (LSB) and 6 (MSB). Only valid if SZI ≠ 0, otherwise node uses 42h.
	43	Used by a node when replying with exactly 4 data bytes. We use only Bytes 5 – 8, LSB first. Only valid if SZI ≠ 0, otherwise node uses 42h.
2	60h, 70h	Used by host. Second step to segmented read process always begins with 60h. Every time the node replies with data, the host toggles between 60h & 70h. If this does not occur, node will abort with code 80h.
	0	Reply from node. Will occur only if host used 60h in the previous command and there is more data to transmit. In this case the host should send another message using 70h in Byte 1, 0 in all other Bytes, to retrieve more data.
	1	Reply from node. Will occur only if host used 60h in the previous command and this message contains the last of the data.
	10	Reply from node. Will occur only if host used 70h in the previous command and there is more data to transmit. In this case the host should send another message using 60h in Byte 1, 00h in all other Bytes, to retrieve more data.
	11	Reply from node. Will only occur if host used 70h in the previous command and this message contains the last of the data.
	3, 5, 7, 9, B, D	Same as 1h except the number of bytes not containing data is specified. Use 3h if last Byte is empty, 5h if last two Bytes are empty, etc. Valid only if SZI ≠ 0, otherwise node will reply with 1h.
	13, 15, 17, 19,	Same as 11h except the number of bytes not containing data is specified. Use 13h if last Byte is empty, 15h if last two Bytes are

	1B, 1D	empty, etc. Valid only if SZI ≠ 0, otherwise node will reply with 1h.

Table 4.1 – Read SDO Byte 1 commands

3. An expedited write SDO is initiated by the host transmitting this message. This is step 1.

COB-ID	600h + Node-ID
Byte 1	22h, 2F, 2B, 23h
Byte 2	Object Index, LSB
Byte 3	Object Index, MSB
Byte 4	Subindex
Byte 5	Data, LSB first
Byte 6	
Byte 7	
Byte 8	

The device at Node-ID responds with this message. This is step 2.

COB-ID	580h + Node-ID
Byte 1	42h, 4Fh, 4Bh, 43h
Byte 2	Object Index, LSB
Byte 3	Object Index, MSB
Byte 4	Subindex
Byte 5	Data, LSB first
Byte 6	
Byte 7	
Byte 8	

4. A segmented write SDO is initiated by the host transmitting this request. This is step 1.

COB-ID	600h + Node-ID
Byte 1	20h, see Table 4.2
Byte 2	Object Index, LSB
Byte 3	Object Index, MSB
Byte 4	Subindex
Byte 5	00h, or number of bytes to transfer, depending upon the mode used.
Byte 6	
Byte 7	
Byte 8	

The device at Node-ID responds with this message. This is still step 1.

COB-ID	580h + Node-ID
Byte 1	60h
Byte 2	Object Index, LSB
Byte 3	Object Index, MSB
Byte 4	Subindex
Byte 5	00h
Byte 6	
Byte 7	
Byte 8	

Then the host begins transferring data. This is now step 2.

COB-ID	600h + Node-ID
Byte 1	0h, 1h, 10h, 11h
Byte 2	Data, LSB first
Byte 3	
Byte 4	
Byte 5	
Byte 6	
Byte 7	
Byte 8	

Then the device replies with:

COB-ID	580h + Node-ID
Byte 1	20h, 30h
Byte 2	NA
Byte 3	
Byte 4	
Byte 5	
Byte 6	
Byte 7	
Byte 8	

The write SDO Byte 1 commands are:

Usage	***Byte 1***	***Meaning***
Host initiates Write SDO with >4 data bytes to send	20h	Used by host when initiating a write transfer of ≥4 data bytes. Total number is not specified. Node replies with 60h, confirming that it is ready to receive data.
	21	Used by host when initiating a write transfer of ≥4 data bytes. Total number of bytes is indicated with Bytes 5 – 8, LSB first. Node replies with 60h, confirming that it is ready. Valid only if SZI ≠ 0, otherwise use 20h.
Host initiates Write SDO with ≤4 data bytes.	22	Used by host when writing ≤4 data bytes. Total number not indicated. Node replies with confirmation 60h.
	2F	Used by host when writing exactly 1 byte, held in Byte 5. Node replies with confirmation 60h. Valid only if SZI ≠ 0, otherwise use 22h.
	2B	Used by host when writing exactly 2 bytes, held in Bytes 5 (LSB) and 6 (MSB). Node replies with confirmation 60h. Valid only if SZI ≠ 0, otherwise use 22h.
	23	Used by host when writing exactly 4 bytes, held in Bytes 5-8, LSB first. Node replies with confirmation 60h. Valid only if SZI ≠ 0, otherwise use 22h.
Data transfer commands	60h 20 30	Reply from node. 60h only occurs once during the initiate write process. Afterward each consecutive reply to a message containing data will toggle between 20h (which must be first) and 30h.
	00	Used by host if the node's previous reply contained 60h or 30h in Byte 1 and there is still data to transmit.
	1	Used by host if the node's previous reply contained 60h or 30h in Byte 1 and this message contains the last data to transfer.
	10	Used by host if the node's previous reply contained 20h in Byte 1 and there is still data to left to transmit.
	11	Used by host if the node's previous reply contained 20h in Byte 1 and this message contains the last data to transfer.
	3, 5, 7, 9, B, D	Same as 1h except the number of bytes not containing data is specified. Use 3h if last byte is empty, 5h if last two are empty, etc. Valid only if SZI ≠ 0, otherwise 1h.

	13, 15, 17, 19, 1B, 1D	Same as 11h except the number of bytes not containing data is specified. Use 13h if last byte is empty, 5h if last two are empty, etc. Valid only if SZI ≠ 0, otherwise 11h.

Table 4.2 – Write SDO Byte 1 commands

SDO Abort Transfer messages

When an error occurs during an SDO the device node is required to transmit this Abort Transfer message:

COB-ID	580h + Node-ID
Byte 1	80h
Byte 2	Object Index, LSB
Byte 3	Object Index, MSB
Byte 4	Subindex
Byte 5	Code from Table 4.3, LSB first
Byte 6	
Byte 7	
Byte 8	

The 4-byte Abort codes are:

Abort code	***Description***
0503 0000h	Toggle bit not alternated
0504 0000	SDO protocol timed out
0504 0001	Command specifier not valid
0504 0002	Invalid block size, block mode only, see DS-301
0504 0003	Invalid sequence number, block mode only, see DS-301
0504 0004	CRC error, block mode only, see DS-301
0504 0005	Out of memory
0601 0000	Unsupported access to an object
0601 0001	Attempt to read a write-only object
0601 0002	Attempt to write a read-only object
0602 0000	Object does not exist in the Object Dictionary
0604 0041	Object cannot be mapped to the PDO
0604 0042	The number and length of the objects to be mapped would exceed the PDO length
0604 0043	General parameter incompatibility reason
0604 0047	General internal incompatibility in the device
0606 0000	Access failed due to hardware error
0607 0010	Data type does not match, length of service parameter does not match
0607 0012	Data type does not match, length of service parameter too high
0607 0013	Data type does not match, length of service parameter too low
0609 0011	Sub-index does not exist
0609 0030	Value range of parameter exceeded (only for write access)
0609 0031	Value of parameter written too high
0609 0032	Value of parameter written too low
0609 0036	Maximum value is less than minimum value
0800 0000	General error
0800 0020	Data cannot be transferred or stored to the application

0800 0021	Data cannot be transferred or stored to the application because of local control
0800 0022	Data cannot be transferred or stored to the application because of present device state
0800 0023	Object Dictionary dynamic generation fails or no OD is present

Table 4.3 – Abort error codes

To repeat, there is some flexibility (and occasional ambiguity) in implementing SDOs. However implemented, though, errors are expected to be detected and reported. The host cannot be led to believe that it has correct information when in fact it does not. A node on a network that throws frequent error codes may need a review of its architecture.

5. Process Data Objects

Using the PDO protocol, we transfer only between 0 and 8 bytes of data, and this may be tailored to suit one's needs. This is much more streamlined than SDOs. Confirmation does not occur. PDOs may be set up in advance by the host, using SDOs, or they may simply be coded into the node's software.

It is possible to set up master-less, peer-to-peer communication this way, although it is the host's responsibility to oversee this and possibly to set it up dynamically. In general, any node may initiate a PDO, and any other node may receive it. There are two types of PDO: TPDOs *transmit* from the device node to the network. RPDOs *receive* data from network to device. In other words, an RPDO is how a device receives needed data.

There is a strict one-to-one mapping of PDO_{node1} to PDO_{node2} (one of which may of course be the host).

Here is an example TPDO from CiA. It is assumed that both nodes have pre-defined the transmitted data content.

Index	***SubIdx***	***Type***	***Description***

1A00h			*TPDO mapping*
	0	Unsigned8	=4, number of following map entries
	1	Unsigned32	= 6000 01 08h (Index 6000, SubIdx 1, 8 bit)
	2	Unsigned32	= 6000 02 08h (Index 6000, SubIdx 2, 8 bit)
	3	Unsigned32	= 6401 01 10h (Index 6401, SubIdx 1, 16 bit)
	4	Unsigned32	= 6401 02 10h (Index 6401, SubIdx 2, 16 bit)
6000			*Process data field, digital inputs*
	0	Unsigned8	=2, number of following subidx entries
	1	Unsigned8	Image of an 8-bit digital input, Din_1
	2	Unsigned8	Image of an 8-bit digital input, Din_2
6401			*Process data field, analog inputs*
	0	Unsigned8	=2, number of following subidx entries
	1	Unsigned16	Image of a 16-bit analog input, Ain_1
	2	Unsigned16	Image of a 16-bit analog input, Ain_2

The actual TPDO looks like this, with a CAN message packet containing 6 used data bytes, 2 of which are unused. Recall, there are always 8 available data bytes in a PDO. An RPDO is similar, although it is framed as a request for data.

COB-ID	280h + Node-ID (typical)
Byte 1	Din_1
Byte 2	Din_2
Byte 3	Ain_1, LSB
Byte 4	Ain_1, MSB
Byte 5	Ain_2, LSB
Byte 6	Ain_2, MSB
Byte 7	Unused
Byte 8	Unused

A TPDO may be set up such that a device issues one automatically every 50 ms, say, or when a threshold value has been detected. The host may issue an RPDO to a device asynchronously whenever it needs to know a status. Again, the TPDOs and RPDOs form pairs, and they occupy unique channels. There is a one-to-one mapping of node to node.

There are four ways to trigger a PDO:

- Event driven. An input device (eg, sensor) may be programmed to respond to a measured parameter change. In such an event it spontaneously transmits the appropriate pre-defined TPDO. Generally these TPDOs have a low numerical value, i.e., high priority, so they tend to get through the network efficiently.
- Time driven. The input device may be programmed to issue a TPDO at known time intervals.
- Individual Polling. CAN bus by itself supports remote request frames. Thus a request may come to any node *A* from any node *B*. CiA

deprecates this practice, as it could easily disrupt CANopen's deterministic addressing system.

- Synchronized. This is a common method used in motion control applications.

In the project under discussion herein we included some simple, 'hard-coded' PDOs for ease of test. Figure 5.1 depicts an RPDO that efficiently writes four different data types. This is synchronized (by the host) every 4 ms.

```
void copRPDO0(canGram *canRxMessage)
{
    static byte daBuff[8] = { 0x34, 0x12, 0xFF, 0xFF};  // 'Reasonable' initializers in case
    byte d;                                             // we Sync write before properly assigning
    unsigned parmUns16;
    extern byte sysFault, controlStatus, ledOnState, sysStatus;
    extern float driveScaleFactor;

    if (sysStatus & SS_SYNCSET)
    {
        controlStatus = CS_CAN_GENERAL;         // Because we've received a valid CAN msg
        parmUns16 = (unsigned) (daBuff[1] << 8) + daBuff[0];    // Little-endian
        driveScaleFactor = ((float) parmUns16) / 65535;
        ledOnState = daBuff[2];
        sysFault &= daBuff[3];                                  // Note AND
        sysStatus &= ~SS_SYNCSET;
    }
    else
    {
        for (d=0; d<8; d++) daBuff[d] = canRxMessage->data[d];
    }
}
```

Figure 5.1 – Code example of simple RPDO

6. Network management

CANopen requires that there be a Network Management (NMT) master. This need not be the host; any device may be so designated. The NMT node supervises the bus, and there are a few different ways it can do so. If it detects a fault condition it takes some remedial response. Whether or not it is responsible for NMT, every device may be expected to participate in network self-test routines. One common test is the heartbeat message. In this protocol, every device issues a heartbeat TPDO, even if ordinarily it only transmits event-driven changes. This is analogous to a 'radio check.'

In CANopen we pre-allocate certain COB-ID ranges for special functions:

Message type	***COB-ID***	***Note***
NMT	0h	Network management, broadcast
Sync	080	Synchronization message, broadcast
Emergency	081-0FF	
Time stamp	100	Broadcast
PDOs	181-57F	
SDOs	581-67F	
NMT error control	701-77F	
Boot-up	701-77F	

Table 6.3 – Preallocated COB-IDs

Every device needs to operate an internal Network Management state machine and server to handle NMT traffic. NMT traffic has the highest possible network priority. CiA DS-301 refers to this server as the Communication State Machine. State transitions are effected by NMT messages.

Figure 6.1 depicts the Communication State Machine as implemented by our project. All CANopen projects must implement something close to this.

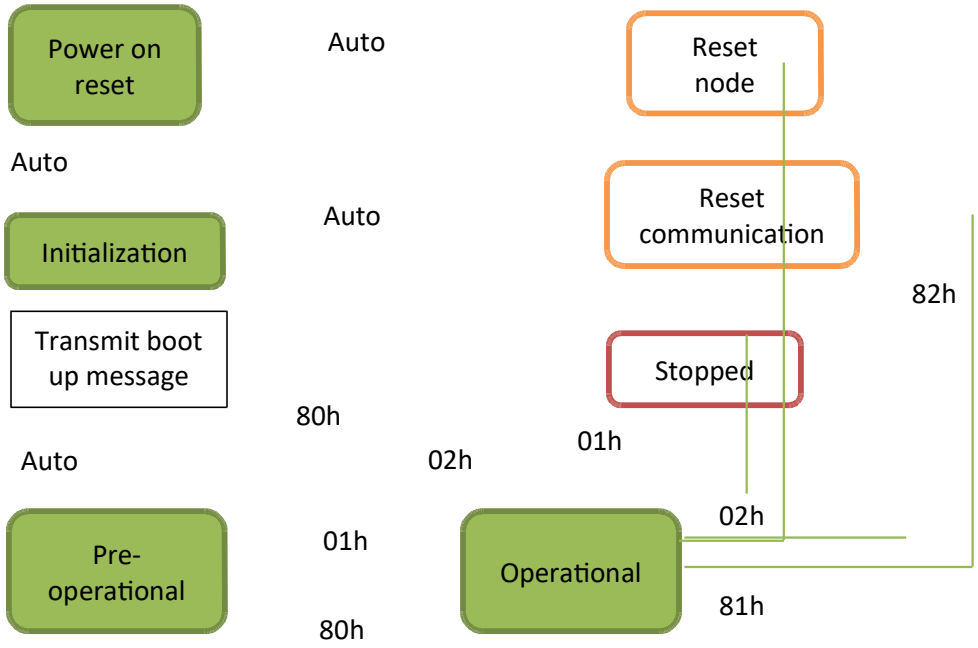

Figure 6.1 – Mandatory Communication State Machine

All NMT messages have COB-ID 000h, RTR = 0. They then have Bytes 1 and 2 as follows:

NMT message	***Byte 1***	***Byte 2***	***Description***
Start remote node	01h	Node ID	Sets state machine at Node ID to Operational. Enable all messages including PDOs.
Stop remote node	02	Node ID	Sets state machine at Node ID to Stopped state. Disable all messages except NMT. Node Guarding / Life Guarding if used are active.
Pre-operational state	80	Node ID	Sets state machine to Pre-operational. In this state only NMT and SDO messages are allowed.
Reset node	81	Node ID	Resets node, same as a power cycle. Node emits Boot Up message

Reset communication	82	Node ID	Reset state machine. Node emits Boot Up message.

Table 6.4 – NMT messages

As mentioned, there are several optional ways to conduct Network Management.

Node Guarding

Node Guarding is a network management routine in which the NMT Master polls each node. The node is required to respond with its communication state within a specified time. The possible responses are as follows. Note, for a given state, the return value is required to toggle between the two values. This assures that the NMT Master that the node is still alive and not stuck in a loop. If the Node Guarding protocol is not observed the NMT Master will take corrective action. The polling periodicity is given by the Guard Time (OD object 100Ch). Response is expected before the occurrence of the next Guard Time.

State Machine	***Return (toggle)***
Stopped	04 / 84h
Operational	05 / 85
Pre-Operational	7F / FF

Table 6.5 – Node Guard responses

The NMT master Node Guard request has COD-ID = 700h + Node-ID, RTR = 1. No other data is transmitted. The response from the host is COB-ID = 700h + Node-ID, and then one byte with the toggling return value from table 6.3.

Life Guarding

A complement to Node Guarding is Life Guarding. Here each device monitors to assure that the NMT master is doing its job. There is an OD object named Life Time Factor (100Dh). Lifetime is defined as the product of Guard Time (object 100Ch) and Life Time Factor. If a node has not received a Node Guard request within this time it should emit a communication error message.

Heartbeat

Another variation on Node Guarding is the heartbeat protocol. Any node may be the designated the periodic heartbeat producer. The other nodes listen for this and regard it as an error condition if they do not hear a heartbeat within the specified time. The producer heartbeat time is OD object 1017h. If this value is 0, that node will not produce a heartbeat. The consumer heartbeat time is OD object 1016h. If this value is 0, that node will not listen for a heartbeat.

The general format for a heartbeat time (object 1016h) is:

Bits 31-24	***Bits 23-16***	***Bits15-0***
Always 00h	Producer Node-ID	Heartbeat time

A host (Node-ID = 0) heartbeat message is simply COB-ID = 700h, RTR = 0, Byte 1 = '00'. If another node is the designated producer, then COB-ID is simply 700h + Node-ID.

Boot-Up

Nodes are required to emit a Boot-Up message after power up, an NMT communication reset, or a software reset. The boot-up message is the same as a heartbeat:
COB-ID = 700h + Node-ID, RTR = 0, Byte 1 = '00'

SYNC messages

SYNC messages are used to simultaneously trigger events in several nodes on the network. Their form is simply COB-ID = 80h, RTR = 0, 0 bytes. Only the host issues SYNCs, and how they are handled by the nodes is application specific.

Emergency messages

Emergency messages are the complement to SYNCs. They may be issued by any node when it detects a serious problem. It is issued only once. The format is:

COB-ID = 80h + Node-ID, RTR = 0,
+ Error Code Bytes

Time Stamps

This is the mechanism for long term system synchronization. The message format is:

COB-ID	100h
Byte 1	Time since midnight, ms, LSB first
Byte 2	
Byte 3	
Byte 4	
Byte 5	Current day since 1 Jan 1984, LSB first
Byte 6	
Byte 7	Unused
Byte 8	Unused

Time stamps need not be periodic. In general though, if they are used, nodes will detect if they have not received a time stamp within some specified period, and this will trigger an error condition.

7. CAN physical and data link layers in more detail

For reference, here are some additional notes on general CAN lower level design.

- We typically use twisted pair cable. The transmission line is typically terminated at each end with ≈120 Ω. Device connections to the two lines must be open drain so that the automatic message prioritization may work.
- Maximum bandwidth is 1Mb/s. Cable distance may extend to ≈40 m at this rate. There are prescribed lower bandwidths with corresponding longer distances.
- Communication is multi-master. It may be initiated at any node, at any time.
- The bit waveform is Non-Return to Zero. There is a bit stuffing requirement to enforce NRZ.
- The maximum latency of a high priority message is < 120 µs (@ 1Mb/s).
- In this project we use CAN 2.0B. While capable of extended, 29 bit messaging we

restrict ourselves to 'Base frame' mode, with 11 bit (SID) message identifiers. We may operate at a bus rate up to 1 Mb/s. CAN 2.0B is defined by ISO 11898.

There are four data frame types that are common to all CAN communication. We give here a brief overview followed by a more detailed analysis.

- Standard Data Frame. This frame is generated by a node when it wishes to transmit data and when it operates in Standard mode with an 11 bit (SID) identifier.

- Extended Data Frame. This frame builds on the Standard frame by adding 18 more bits (the EID) to the identifier.

- Remote Frame. As a peer-to-peer network, node *A* may request a data exchange with node *B*. Remote frames encode this request. This is an essential CAN situation. Our module may be either the *A* or *B* node, and it may be the destination or the source. The destination node sends a remote frame with an identifier that matches that of the required data frame. Recall, in a CAN network, every datum (object) has a unique address. The data source node then transmits the appropriate data frame.

- Error Frame. This is generated by any node that detects a bus error. It has two fields: an error flag field and an error delimiter field.

- Overload Frame. Two conditions may cause a node to precipitate this frame.
  a. The node detects a dominant (0) bit during interframe space. This is an illegal condition.
  b. The node is not ready to start reception of the next message due to some system constraint. A node may generate up to two sequential overload frames in order to delay the start of the next message.

There is also a defined Interframe Space. This refers to the temporal gap between the end of a frame of any sort, and the beginning of a Data or Remote Frame. The CAN standard specifies a hold duration here.

The Standard Data Frame message package has this format:

Name	***Length, bits***	***Constraint***	***Field***
SOF, Start of Frame	1		
Message ID, 'COB-ID' or SID	11		Arbitration, inherently defines priority
RTR, Remote Transmission Request	1	Dominant 0 usually	
IDE, Identifier Extension	1	Dominant 0 for Standard Frame	Control
Reserved (Microchip RB0)	1	Dominant 0	
DLC, Data Length Code. Indicates number of data bytes.	4	0 – 8 bytes	
Data	0 – 64, as set		Data

	by DLC		
CRC, Cyclic Redundancy Check	15		CRC
CRC Delimiter	1	Recessive 1	
ACK Slot	1	Transmitter sens Recessive 1 and any receiver can assert a Dominant 0	ACK
ACK Delimiter	1	Recessive 1	
EOF, End of Frame	7	Recessive 1	
IFS, Interframe Space	3	Recessive 1s. Not part of frame per se, but required by CAN.	

Table 7.6 – Standard CAN message package

Some sections of the frame are referred to as fields. The Arbitration field, for example, is the address that inherently arbitrates priority for the message.

The Extended Data Frame message package has this format:

Name	***Length, bits***	***Constraint***	***Field***
SOF, Start of Frame	1		
Message ID, 'COB-ID' or SID	11		Arbitration, inherently defines priority
SRR, Substitute Remote Request	1	Dominant 0	
IDE, Identifier Extension	1	Recessive for Extended Frame	
EID, Extended Identifier	18		
RTR, Remote Transmission Request	1	Dominant 0	Control
Reserved (Microchip RB1)	1	Dominant 0	
Reserved (Microchip RB0)	1	Dominant 0	
DLC, Data Length Code. Indicates number of data bytes.	4	0 – 8 bytes	
Data	0 – 64, as set by DLC		Data
CRC, Cyclic Redundancy Check	15		CRC
CRC Delimiter	1	Recessive 1	
ACK Slot	1	Transmitter sends Recessive	ACK

		1 and any receiver can assert a Dominant 0	
ACK Delimiter	1	Recessive 1	
EOF, End of Frame	7	Recessive 1	
IFS, Interframe Space	3	Recessive 1s. Not part of frame per se, but required by CAN.	

Table 7.7 – Extended CAN message package

A Remote Frame follows the same format as the Standard or Extended Frames. However, we set RTR to recessive 1 and we set DLC to 0. Since there is no Data exchanged the data field length is 0.

An Error Frame may be generated by any node on the network that detects a bus error. There is a 6 bit Error Flag field, followed by the Error Delimiter, which is always 8 recessive 1's. The Error Flag field may be:

a. Active. 6 dominant 0's. This forces all other nodes to generate Error Echo Flags. This will result in a series of 6 – 12 dominant 0's in the bus.
b. Passive. 6 recessive 1's. This will not affect any other nodes. It does signal to the transmitter that the communication was unsuccessful.

As a node transmits a message, which it may do at any time, it monitors the bus. If it detects that a more dominant message (one with a lower numerical ID) then there is a bus collision and it should stop transmitting. In theory however the highest priority message will punch through. The node that has detected a collision and has paused should reattempt after some random delay time.

Drivers & Isolation

While it can lead to latency problems of its own, galvanic isolation is often recommended in order to protect the embedded electronics from undesired current flow paths. In a typical industrial setting, with motors, transients, ground loops, variable ground potentials, etc., signaling may easily be corrupted. Indeed, it is quite possible to damage embedded electronics this way. Just to pick one source, Analog Devices® makes an excellent line of signal line isolators.

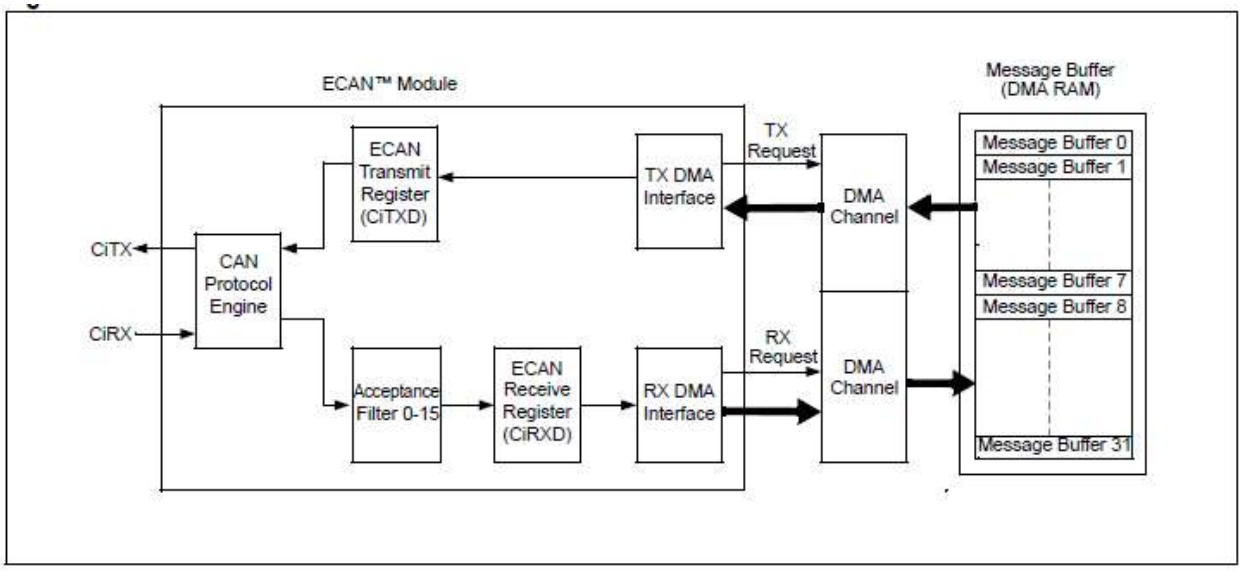

Figure 8.1 – Microchip ECAN architecture

8. Microchip dsPIC33F specifics

The project CAN communication physical layer is based on a Microchip dsPIC33F, which has an 'ECAN' module. By this, Microchip means that it has Enhanced CAN capability, over and above some earlier offerings. Figure 8.1 depicts the architecture of Microchip ECAN. Some salient features are:

- Any level of CAN protocol up to 2.0B is possible. In this project we use Standard frames and a maximum possible bit rate of 1 Mb/s.
- Automatic response to remote transmission requests.
- Up to 8 transmit buffers with application specified priority and abort capability. Each buffer may contain up to eight bytes of data.
- Up to 32 receive buffers. Each buffer may contain up to eight bytes of data.
- Up to 16 full (SID or EID) acceptance filters.
- 3 full acceptance filter masks.
- DeviceNet addressing. This is not used in current project but useful elsewhere.

The dsPIC33F has six operational modes vis à vis the CAN network. Before switching modes there is typically a constraint in which the ECAN module waits for an idle bus condition (typically ≥ 11 recessive bits in series). The six modes are:

- Initialization Mode. Transmission and reception inhibited. Error counters are cleared and interrupts disabled. Configuration registers are accessible only in this mode. CAN protocol will not be violated.
- Disable Mode. Transmission and reception inhibited. ECAN may set the WAKIF interrupt flag bit due to bus activity. Pending interrupts remain and error counter holds its value.
- Normal Operation Mode. Ordinary CAN participation.

- Listen Only Mode. This effectively disables CAN functionality, but it is possible to monitor CAN bus traffic and ascertain the bit rate.
- Listen All Messages Mode. Here we ignore all errors and receive any message. The data which are in the message assembly buffer, up until the occurrence of an error, are copied into the receive buffer and may be examined by the CPU.
- Loopback Mode. Here we exercise all CAN functionality except for the actual physical CAN bus transmission and reception.

The Protocol Engine assures compliance with CAN 2.0B. The Acceptance Filters examine each received message to see if it should be transferred via DMA to the Message Buffer. Upon receipt of a message into the C1RXD register, the receive DMA interface generates an interrupt, and this initiates the DMA transfer. When the module wants to transmit a message, the TX DMA interface generates an interrupt to start the DMA transfer, via C1TXD register. In the Message Buffer array, messages 0-7 can be configured for either transmit or receive operation. Messages 8-31 are receive-only.

9. PI control real-time performance

The application requires that we update the eight analog outputs every 4 ms via a Sync RPDO from the host. In that RPDO is coded the desired current level for the eight outputs. Each of the eight devices responds quite differently and all are remarkably nonlinear. They are subject to thermal runaway and self-destruction. Therefore it is necessary to monitor them in software, using the numerous 12-bit ADCs available on the dsPIC33F.

We deemed it advantageous to use floating point math, as this made calculations more lucid for the customer, as opposed to cryptic (albeit more efficient) fixed point routines. Even so, we were able to meet the real-time constraint. The dsPIC33F was running on its internal PLL, yielding 40 MPS.

For CAN traffic we set up three DMA receive buffers and one transmit DMA buffer. This seems adequate. CAN traffic was assigned the highest priority interrupt, higher than the ADC and Real-time (RTC) mechanism. The analog sample functions disable other interrupts while they are executing, and a fair amount of sampling and processing were found to be necessary. The RTC interrupt ISR is atomic. It uses a semaphore mechanism which tolerates an acceptable amount of jitter.

The familiar Proportional + Integral method of control proved adequate, even for these unstable, nonlinear devices. A layer of robust 'meta-control' supervises the algorithm. PI coefficients were kept sub-optimal by a large degree, but this was acceptable in terms of their actual intended use in the customer process. The coefficient values were derived from a custom Matlab model.

It is not obvious how one should handle exceptions in a CANopen network. In this instance, many of the self-check measurements fail briefly, then autoclear. In this application,

CANopen is the only practical way for the user to interface with our control module. In development we supported a highly developed diagnostic serial output. This helps immensely, but is not practicable in final use.

Should transient measurement errors throw CANopen faults? This is a difficult call. We elected to support a priority schedule, which is easily modifiable in firmware. Any exceptions that exceed a threshold level get reported back to the host as CANopen errors.

10. CAN / CANopen glossary

- ARQ – Automatic Repeat Request
- COB – Communication Object. This is a basic CAN data package. Every CAN network allows 10^{11} = 2048 COBs, each of which may contain up to 8 bytes of data. Every COB has a unique ID. This inherently determines the priority of the COB in the MAC sub-layer.
- COB-ID – Connection Object Identifier. There is some ambiguity in this term between Advanced Motion Controls and CiA. AMC simply defines COB-ID as the 11 bit CAN address. CiA defines COB-ID as the dedicated link, or address space or channel, of RPDOM1 in the host to TPDOX1 in device X. A COB-ID also may contain configuration bits, such as a PDO enable bit. For the most part we follow AMC herein.
- CRC – Cyclic Redundancy Check.
- CSDO – Client SDO.
- DLC – Data Length Code. In the basic CAN message the DLC indicates the number of data bytes to follow. This field is 4 bits but is limited between 0 and 8 data bytes. This was an early Bosch design decision minimize message throughput times.
- ECAN – Microchip terminology for Enhanced CAN support.
- EDS – Electronic Data Sheet. A document published by a device manufacturer that specifies its Device Profile. This is a data structure that may be read as part of the device's Object Dictionary that contains any parameters of interest for the device. The format of the EDS may be in a type of Microsoft '.ini' form or in XML.
- EID – Extended Identifier, an additional 18 bits, to total 29 bits in extended mode addressing.
- LLC – Logical Link Control. A sub-layer of the Data Link Layer in the CAN stack that gives the user an interface that is independent of the underlying MAC layer.
- MAC – Medium Access Control. A sub-layer of the Data Link Layer in the CAN stack that controls who gets access to the medium in order to send a message.
- NMT – Network Management. One of the service elements of the application layer in the CAN stack. The NMT configures, initializes, and handles errors in CAN network.

- Node-ID. In CANopen every device or node is assigned a unique ID, between 1 – 127. Node-ID = 0 is reserved for the host.
- OD – Object Dictionary.
- PDO – Process Data Object.
- Remote COB. A COB whose transmission can be requested by another device.
- RPDO – Receive PDO.
- SDO – Service Data Object.
- SID – Standard Identifier, 11 bit.
- SSDO – Server SDO.
- SYNC – Synchronization Object.
- SZI – An Object Dictionary datum at location 2111h, the Size Indicator holds the number of bytes to transfer.
- TPDO – Transmit PDO.

11. References

- Advanced Motion Controls, Camarillo, CA, *CANopen Communication Reference Manual*.
- *CiA Draft Standard 301, CANopen Application Layer and Communication Profile, Version 4.02*, February 2002.
- Copley Controls, *CANopen Programmer's Manual*, rev 5, Oct 2008.
- Fosler, Ross M, Microchip Technology, *AN 925: A CANopen Stack for PIC18 ECA Microcontrollers*, 2004.
- Microchip Technology, *dsPIC33FJXXXGP Data Sheet*, 2009.
- Microchip Technology, *dsPIC33F Family Reference Manual, Section 21, ECAN*, 2007.
- Pfeiffer, Olaf, et al, *Embedded Networking with CAN and CANopen*, Copper Hill Media, 2008.