INTRODUCTION

CANopen is a field bus protocol based on the Controller Area Network (CAN). As the name implies, it is an open network standard accepted throughout the world. While created as a field bus protocol for industrial automation, CANopen finds use in a wide range of other non-industrial applications. There are so many possibilities, in fact, that it is possible to write volumes on specialized uses of the protocol.

Rather than being specific to one narrow application or even one field, we present here a more generalized approach: a generic communication stack based on CANopen that can be tailored to the user’s needs. This article focuses only on what is covered in the CAN in Automation (CiA) standard DS-301. In fact, most of the discussion is limited to the predefined areas of the specification, with emphasis on understanding how the code provided with this application note functions and how users might develop an application on the CANopen Stack. To help illustrate this, a simple example application is developed based on the CiA DS-401 specification, Generic I/O Modules. The additional code provided is solely for demonstration; thus there is no detailed discussion of the demonstration code. However, code examples with comments from the demo application are frequently used throughout this document.

All code provided with this application note is developed for the PIC18F8680 and PIC18F4680 families of devices, which include ECAN technology as part of their peripheral set. It is designed to compile with Microchip’s C18 v2.30 (or greater) compiler. Although developed for these specific device families, the code is adaptable to other PIC18 families with CAN.

It is expected that the reader already has some knowledge of CANopen, or has access to the latest CANopen standard (listed in the References section) to refer to for theory and/or critical terminology. The information covered in this application note leans towards understanding the implementation and developing on that foundation, rather than discussing the many details of CANopen.

OVERVIEW OF THE STACK

The CANopen Stack provides the lower layers of the protocol. Some of the features of this design include:

- Embedded state machine for handling all communications between all nodes and objects
- Default Service Data Object (SDO) Server
- Up to 4 transmit and 4 receive Process Data Objects (TPDOs and RPDOs)
- Explicit and Segmented Messaging Support
- Statically-mapped PDO support
- Structured dictionary for the PDOs and SDO
- Node Guard/Life Guard
- SYNC consumer
- Heartbeat Producer
- ECAN Driver support

As this list shows, the CANopen Stack discussed here is designed for applications that are typically more “slave”. This design is more static in nature, which leads to more efficient code with better effective use of code space.

In addition, the actual CANopen code is broken into a series of smaller source and header files, all written in C. This allows users to select the appropriate services that they may need for their application and selectively build a project tailored to their specific requirements. A complete list of source files is presented in Table 1.

Of course, the actual application and some aspects of the communications must still be developed by the user. The provided CANopen Stack code affords a base on which the application may be built.
## TABLE 1: CANopen SOURCE FILES

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO_CANDRV.c</td>
<td>ECAN module driver. These files may be replaced by other device-specific drivers, if required.</td>
</tr>
<tr>
<td>CO_CANDRV.h</td>
<td></td>
</tr>
<tr>
<td>CO_COMM.c</td>
<td>Communications management services. Required for all applications.</td>
</tr>
<tr>
<td>CO_COMM.h</td>
<td></td>
</tr>
<tr>
<td>CO_DEV.c</td>
<td>Device specific files. Users must edit this file for their device.</td>
</tr>
<tr>
<td>CO_DEV.h</td>
<td></td>
</tr>
<tr>
<td>CO_DICT.c</td>
<td>The object dictionary. Required for all applications.</td>
</tr>
<tr>
<td>CO_DICT.h</td>
<td></td>
</tr>
<tr>
<td>CO_DICT.def</td>
<td></td>
</tr>
<tr>
<td>CO_MAIN.c</td>
<td>CANopen main services. Required for all applications.</td>
</tr>
<tr>
<td>CO_MAIN.h</td>
<td></td>
</tr>
<tr>
<td>CO_MEMIO.c</td>
<td>Memory copy functions used by the dictionary. Required for all applications.</td>
</tr>
<tr>
<td>CO_MEMIO.h</td>
<td></td>
</tr>
<tr>
<td>CO_NMT.c</td>
<td>Network management communications endpoint.</td>
</tr>
<tr>
<td>CO_NMT.h</td>
<td></td>
</tr>
<tr>
<td>CO_NMTE.c</td>
<td>Node Guard, Heartbeat and Boot-up communications endpoint.</td>
</tr>
<tr>
<td>CO_NMTE.h</td>
<td></td>
</tr>
<tr>
<td>CO_PDO.c</td>
<td>General PDO services.</td>
</tr>
<tr>
<td>CO_PDO.h</td>
<td></td>
</tr>
<tr>
<td>CO_PDO1.c</td>
<td>PDO object handling endpoints. Provided in a template format that requires development by the user for the specific application. Must be used with the general PDO services files.</td>
</tr>
<tr>
<td>CO_PDO1.h</td>
<td></td>
</tr>
<tr>
<td>CO_PDO2.c</td>
<td></td>
</tr>
<tr>
<td>CO_PDO2.h</td>
<td></td>
</tr>
<tr>
<td>CO_PDO3.c</td>
<td></td>
</tr>
<tr>
<td>CO_PDO3.h</td>
<td></td>
</tr>
<tr>
<td>CO_PDO4.c</td>
<td></td>
</tr>
<tr>
<td>CO_PDO4.h</td>
<td></td>
</tr>
<tr>
<td>CO_SDO1.c</td>
<td>Default server SDO communications endpoint.</td>
</tr>
<tr>
<td>CO_SDO1.h</td>
<td></td>
</tr>
<tr>
<td>CO_SYNC.c</td>
<td>Consumer synchronization communications endpoint.</td>
</tr>
<tr>
<td>CO_SYNC.h</td>
<td></td>
</tr>
<tr>
<td>CO_TOOLS.c</td>
<td>Tools for converting Microchip and CANopen CAN identifier formats. For better process performance, all COB IDs are stored internally in the Microchip format. When COB ID is presented due to a request, then the ID is converted to CANopen.</td>
</tr>
<tr>
<td>CO_TOOLS.h</td>
<td></td>
</tr>
<tr>
<td>CO_ABERR.h</td>
<td>Common error definitions. Required for all applications.</td>
</tr>
</tbody>
</table>
**CANopen FIRMWARE MODEL**

The firmware is designed in three levels, as shown in Figure 1. The lowest level is the ECAN driver providing hardware abstracted CAN support. The communications management level is the primary interface between the driver and the individual endpoint handling.

Besides the application, there is also the dictionary. In essence, it resides outside of the communication object, and is directly connected to the SDO endpoint.

**The Driver**

At the lowest level is the ECAN driver, which serves as an abstracted hardware interface. It is implemented by the source files `CO_CANDRV.c` and `CO_CANDRV.h`.

The driver handles all ECAN hardware related functionality, and conveniently abstracts much of the complex filtering that is part of the CAN protocol. This is discussed in greater detail later in this document.

**Communications Management**

The communications manager is part of the total communications object. It is provided to capture any events from the ECAN driver and the higher application levels, and dispatch these to the appropriate handling communications sub-objects and functions. Essentially, opening, closing, transmitting to, and receiving from an endpoint is all directed by the communications manager. Communications management is provided in the files `CO_COMM.c` and `CO_COMM.h`.

The manager has knowledge of what state each endpoint is in as well as the state of the device globally. Thus it can block messages to endpoints as necessary based on local or global state.

Another feature of the manager is that it uses a single-byte “handle” method supported by the driver to decode message events. The handle is of a particular structure designed to accelerate performance; it is significantly faster than decoding the 11-bit or 29-bit CAN identifier in order to determine the handling function for a particular message.

**FIGURE 1: BASIC FIRMWARE MODEL OF THE CANopen STACK**

![Diagram of CANopen Firmware Model](image-url)
Endpoints

The CANopen specification defines several possible endpoints. The five endpoint objects listed below are implemented in this example; others may be made available in the future.
- The Default Server SDO
- Up to four Static PDOs
- Synchronization Consumer
- Network Management Slave
- Node Guard or Heartbeat

SERVER SDO COMMUNICATION

The default server SDO (Service Data Object) is provided. The SDO communications path is directly linked to the object dictionary; SDO messages contain information that relates the SDO to a particular object. Data in every message is decoded, validated, and (if valid) eventually executed.

There are essentially two basic operations: read and write. Thus each complete SDO transfer (which may be multiple messages) will either read or write a single object referenced in the dictionary. The default SDO is contained in the source files CO_SDO1.c and CO_SDO1.h.

PDO COMMUNICATION

The PDO (Process Data Object) communications path is linked directly to the applicable application object or objects. Thus the path is assumed by the device and no path information is contained within the communication. Essentially the data is mapped internally to one or more objects. Data is either statically mapped (compiled) or dynamically mapped (set at runtime). One message can contain data from more than one object.

The firmware provided with this application note supports the four default PDOs. Overall PDO services are provided in the source files CO_PDO.c and CO_PDO.h. The additional files CO_PDO1.c and CO_PDO1.h (where n may have a value of 1 to 4) are used to implement the individual PDOs. These are provided in template form, and must be developed to meet the application requirements.

NETWORK MANAGEMENT CONSUMER

A Network Management (NMT) slave is provided as required by the specification. The NMT Object receives commands to change the state of the device or reset the device’s application and/or communications. Figure 2 shows the CANopen state machine, as well as the commands that trigger state changes.

Network management is provided in the source files CO_NMT.c and CO_NMT.h.

FIGURE 2: STATE MACHINE FOR A CANopen DEVICE

Note: Unlabeled transitions (shown with darker lines) are automatic and do not require an external event.
NODE GUARD/HEARTBEAT
There is a single Node Guard or Heartbeat endpoint as required by the CANopen specifications. They both exist in code; however, only one of these Watchdog methods are enabled at any given time (also defined in the specifications).

Node Guard and Heartbeat endpoint functionality is provided in the source files CO_NMTE.c and CO_NMTE.h.

SYNCHRONIZATION CONSUMER
One synchronization consumer (SYNC) is provided. The SYNC message is simply an event to the application to generate any synchronized PDO messages.

The source files CO_SYNC.c and CO_SYNC.h contain the SYNC object.

The Dictionary
The object dictionary functions as a central information database for the device. Every object within the device is represented within the dictionary by an index, sub-index, and some access information. An object can be as simple as a single byte of data or a more complex data structure. Table 2 shows the basic areas of the dictionary that are defined by index in the CANopen specification.

The development and definition of dictionary objects is discussed in greater detail in “Objects and the Object Dictionary” (page 36).

TABLE 2: LOCATION RANGES WITHIN THE OBJECT DICTIONARY

<table>
<thead>
<tr>
<th>Index</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001-001F</td>
<td>Static Data Type</td>
</tr>
<tr>
<td>0020-003F</td>
<td>Complex Data Types</td>
</tr>
<tr>
<td>0040-005F</td>
<td>Manufacturer Specific Data Types</td>
</tr>
<tr>
<td>0060-007F</td>
<td>Device Profile Static Data Types</td>
</tr>
<tr>
<td>0080-009F</td>
<td>Device Profile Complex Data Types</td>
</tr>
<tr>
<td>00A0-0FFF</td>
<td>Reserved</td>
</tr>
<tr>
<td>1000-1FFF</td>
<td>Communication Profile Area</td>
</tr>
<tr>
<td>2000-5FFF</td>
<td>Manufacturer Specific Profile Area</td>
</tr>
<tr>
<td>6000-9FFF</td>
<td>Standardized Profile Area</td>
</tr>
<tr>
<td>A000-FFFF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

By using the index, any defined object can be accessed. From the network point of view, access to an object is provided through the SDO or PDO endpoint as shown in Figure 1. CANopen dictionary functionality is implemented with these files:
- CO_DICT.c
- CO_DICT.h
- CO_DICT.def
- CO_STD.def
- CO_MFTR.def
- CO_PDO.def

Standard Device Objects
The standard device objects, although not shown in Figure 1, are required by the specification. The standard objects include information such as status, the device name, serial number, and version information. They are provided in the source files CO_DEV.c and CO_DEV.h.

Application Objects
At the upper level of the stack is the application object, which must be defined for the specific application and included in the dictionary. The actual objects are defined and written by users for their specific application.

Other Firmware
There are other files provided to define standard data types, define errors, support memory copy functions, and supply COB ID conversion tools. They are:
- CO_TOOLS.c
- CO_TOOLS.h
- CO_MEMIO.c
- CO_MEMIO.h
- CO_ABERR.h
- CO_TYPES.h
COMPILE TIME SETUP

There are a total of 40 compile time options available to configure the source code for a particular application. Most of these are used to configure the factors that control the CAN bit rate (Phase Segment timing, Synchronization Jump Width, baud rate prescaler, etc.). All of the options are listed in Table 3.

Setting Device Information

The CANopen specification identifies a number of objects that identify a particular device. Device specific information is provided through a simple set of data that is referenced from the object dictionary. This information must be included in developing the application. Table 4 lists these objects.

<table>
<thead>
<tr>
<th>Table 3: Compile Time Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>CAN_BITRATE0_BRGCON1</td>
</tr>
<tr>
<td>CAN_BITRATE0_BRGCON2</td>
</tr>
<tr>
<td>CAN_BITRATE0_BRGCON3</td>
</tr>
<tr>
<td>CAN_BITRATE1_BRGCON1</td>
</tr>
<tr>
<td>CAN_BITRATE1_BRGCON2</td>
</tr>
<tr>
<td>CAN_BITRATE1_BRGCON3</td>
</tr>
<tr>
<td>CAN_BITRATE1</td>
</tr>
<tr>
<td>CAN_MAX_RCV_ENDP</td>
</tr>
<tr>
<td>CO_NUM_OF(PDO)</td>
</tr>
<tr>
<td>CO_SPEED_UP_CODE</td>
</tr>
<tr>
<td>CO_SDO1_MAX_RX_BUF</td>
</tr>
<tr>
<td>CO_SDO1_MAX_SEG_TIME</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4: Standard Device Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Name</td>
</tr>
<tr>
<td>rom unsigned long rCO_DevType</td>
</tr>
<tr>
<td>rom unsigned char rCO_DevName[]</td>
</tr>
<tr>
<td>rom unsigned char rCO_DevHardwareVer[]</td>
</tr>
<tr>
<td>rom unsigned char rCO_DevSoftwareVer[]</td>
</tr>
<tr>
<td>rom unsigned char rCO_DevIdentityIndex</td>
</tr>
<tr>
<td>rom unsigned long rCO_DevVendorID</td>
</tr>
<tr>
<td>rom unsigned long rCO_DevProductCode</td>
</tr>
<tr>
<td>rom unsigned long rCO_DevRevNo</td>
</tr>
<tr>
<td>rom unsigned long rCO_DevSerialNo</td>
</tr>
<tr>
<td>unsigned char uCO_DevErrReg</td>
</tr>
<tr>
<td>unsigned long uCO_DevManufacturerStatReg</td>
</tr>
</tbody>
</table>
WRITING THE APPLICATION

There is significant work that goes into developing an application and communications according to the CANopen specifications. The firmware provided eliminates some of the effort by providing some of the lower-level communications handling. Aside from the work necessary to develop the application itself, the following items must be developed for the application.

- Define the application objects in the dictionary
- Develop handling for complex objects
- Develop handling functions for the necessary CANopen communications events
- Develop PDOs

This section introduces the “toolbox” provided by the associated firmware. All the event functions and services are described for any application that may need them.

Main Services

The CANopen protocol is started by calling the mCO_InitAll() function. This issues a CAN driver Reset and causes the boot-up message to be sent. However, prior to starting the CANopen protocol, the default communications specific parameters must be set to their appropriate state. For example, the node_id and baud rate are critical for proper messaging. Other settings include the Node Guard settings, Heartbeat settings, the device error object, as well as the manufacturer specific status.

Once started, all processing occurs through the functions mCO_ProcessAllEvents() and mCO_ProcessAllTimeEvents(). The first handles all general communications related processing like sending and receiving CAN messages for each endpoint. The later function handles communication endpoints that have specific time requirements such as the NMTE (Heartbeat/Node Guard) and any PDO endpoint. The mCO_ProcessAllEvents() function should be called as often as possible to capture all messaging events from the driver. The mCO_ProcessAllTimeEvents() function should be called at 1 ms intervals.

mCO_ProcessAllEvents

This is the main routine from which all events are processed. From this, transmit and receive events are processed within the Communications Manager. This function must be called as often as possible to process any communications events. How often this needs to be called is highly dependent on the driver and the necessity to respond to driver events before overflow.

Syntax

void mCO_ProcessAllEvents(void)

Parameters

None

Return Values

None

Example

(See following page)
Example

```c
void main(void)
{
    // Perform any application specific initialization
    TimerInit(); // Init my timer

    mSYNC_SetCOBID(0x12); // Set the SYNC COB ID (MCHP format)
    mCO_SetNodeID(0x01);  // Set the node_id
    mCO_SetBaud(0x00);    // Set the baudrate
    mNMT_SetHeartBeat(0x00); // Set the initial heartbeat
    mNMT_SetGuardTime(0x00); // Set the initial guard time
    mNMT_SetLifeFactor(0x00); // Set the initial life time
    mCO_InitAll(); // Initialize CANopen to run

    while(1)
    {
        // Process CANopen events
        mCO_ProcessAllEvents();
        // Process application specific functions
        // 1ms timer events
        if (TimerIsOverflowEvent())
        {
            // Process timer related events
            mCO_ProcessAllTimeEvents();

            // Perform other time functions
        }
    }
}
```
mCO_ProcessAllTimeEvents
This is the main routine from which all low-resolution time-related events are processed. This function must be called every 1 ms. High-resolution events (typically in the µs region) must be handled in the application. Internally this function ensures that all objects in the stack that require time control get a tick event.

Syntax
void mCO_ProcessAllTimeEvents(void)

Parameters
None

Return Values
None

Example
Refer to the example provided in mCO_ProcessAllEvents.

mCO_InitAll
This function must be called after setting up all initial object parameters. It will issue a Reset to the CAN driver and start opening the required communications. Once called, the node will be live on the network and the boot-up message will be sent.

Syntax
void mCO_InitAll(void)

Parameters
None

Return Values
None

Example
Refer to the example provided in mCO_ProcessAllEvents.

mCO_SetNodeID
Call this function to set the node_id. node_id must be an unsigned char with the Most Significant bit reserved. In addition, the CANopen specifications reserve the NodeID 00h; valid values for the NodeID range from 01h to 7Fh. This function must be called prior to mCO_InitAll() to effectively set the ID.

Syntax
void mCO_SetNodeID(unsigned char node_id)

Parameters
unsigned char node_id: The node_id for this node, valid range from 01h to 7Fh.

Return Values
None

Example
Refer to the example provided in mCO_ProcessAllEvents.
**mCO_GetNodeID**

Call this function to get the current ID used by the stack. The ID is returned as an unsigned char.

**Syntax**

```c
unsigned char node_id mCO_GetNodeID(void)
```

**Parameters**

None

**Return Values**

`unsigned char node_id:` The node_id for this node, valid range from 01h to 7Fh.

**Example**

None

**mCO_SetBaud**

Call this function to set the baud rate of the node. The value must be between 0 and 8 inclusive. Any other value will default to the 0 setting. The exact baud rate is determined by the CAN driver definitions (page 46). This function must be called prior to `mCO_InitAll()` to change the baud rate.

**Syntax**

```c
void mCO_SetBaud(unsigned char bitrate)
```

**Parameters**

`unsigned char bitrate`

**Return Values**

None

**Example**

Refer to the example provided in `mCO_ProcessAllEvents`.

**mCO_GetBaud**

Call this function to get the current baud rate used by this node. The baud rate is returned as an unsigned char. The exact baud rate is determined by the CAN driver definitions (see “ECAN™ Driver”, page 46).

**Syntax**

```c
unsigned char mCO_SetBaud(void)
```

**Parameters**

None

**Return Values**

`unsigned char:` The current bit rate setting used by the node.

**Example**

Refer to the example provided in `mCO_ProcessAllEvents`.
PDO Events and Services

This section describes the functions used for PDO support. All of these are essentially low-level communications support such as opening, closing, and communicating with specific PDO endpoints. Before discussing these functions, however, a review of how to develop these data objects is in order.

PDO DEVELOPMENT

A critical part of the application design task is developing PDOs. Some decisions have to be made regarding what features to support: choosing between dynamic and static PDO mapping, selecting a Transmission Synchronization mode, and whether or not to support inhibit time. The CANopen Stack source code provided includes a base set of tools to support PDO communication for which such features can be built on.

The critical points for developing PDO support includes developing code to handle these items:
- PDO Communications events
- PDO Mapping
- PDO Synchronization
- PDO Event and Inhibit time

PDO Communications Events

Every enabled PDO will have some communications events to support setting the typical aspects of the PDO. Events are actually call back functions specified in the dictionary to handle specific PDO communications parameters. For example, a master sends a request via an SDO to a slave device to change the type of the PDO (refer to the specifications for information on communication types). The request is passed upwards through the stack to the dictionary and eventually to the function that handles access to the type.

Example 1 and Example 2 demonstrate the link between the dictionary and the actual function _CO_COMM_TPDO1_TypeAccessEvent_. Example 1 shows the entry in the dictionary. Example 2 shows the actual callback. In this case the example demonstrates support only for types 0 to 240, 254, and 255. (The PDO transmission types are shown in Table 5.) Note that none of the events are discussed in detail since they are created by the application designer and thus, handled by the designer’s firmware.

**EXAMPLE 1: PDO DICTIONARY ENTRY**

```plaintext
{0x1800,0x00,CONST,1,{(rom unsigned char *)&uDemoTPDO1Len}},
{0x1800,0x01,RW | FUNC,4,{(rom unsigned char *)&CO_COMM_TPDO1_COBIDAccessEvent}},
{0x1800,0x02,RW | FUNC,1,{(rom unsigned char *)&CO_COMM_TPDO1_TypeAccessEvent}}
```
EXAMPLE 2: EVENT HANDLER

void CO_COMM_TPDO1_TypeAccessEvent(void)
{
    unsigned char tempType;
    switch (mCO_DictGetCmd())
    {
        //case DICT_OBJ_INFO:// Get information about the object
        // The application should use this to load the
        // structure with length, access, and mapping.
        // break;
    
    case DICT_OBJ_READ: // Read the object
        // Write the Type to the buffer
        *(uDict.obj->pReqBuf) = uDemoSyncSet;
        break;

    case DICT_OBJ_WRITE: // Write the object
        tempType = *(uDict.obj->pReqBuf);
        if ((tempType >= 0) && (tempType <= 240))
        {
            // Set the new type and resync
            uDemoSyncCount = uDemoSyncSet = tempType;
        }
        else
        if ((tempType == 254) || (tempType == 255))
        {
            uDemoSyncSet = tempType;
        }
        else {mCO_DictSetRet(E_PARAM_RANGE);} //error
        break;
    }
}

TABLE 5: PDO TRANSMISSION TYPES

<table>
<thead>
<tr>
<th>Transmission Type</th>
<th>PDO Transmission Sync Character</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cyclic</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1 through 240</td>
<td>X</td>
</tr>
<tr>
<td>241 through 251</td>
<td></td>
</tr>
<tr>
<td>252</td>
<td>X</td>
</tr>
<tr>
<td>253</td>
<td></td>
</tr>
<tr>
<td>254</td>
<td></td>
</tr>
<tr>
<td>255</td>
<td></td>
</tr>
</tbody>
</table>

PDO Mapping

PDO mapping can be either static or dynamic. No code is provided specifically for support for either. However, no code is really necessary to represent static mapping. Thus, static code is significantly easier and requires less processing to support. Dynamic PDO mapping is more challenging because it requires referencing the dictionary one or multiple times per PDO. Only static mapping is demonstrated for this version of the CANopen Stack.

Example 3 shows the entry within the dictionary. The actual mapping is just ROM data as shown in Example 4. Any requests through the default SDO to the mapping data in the dictionary will read static data directly from ROM. It is assumed that the static data stored in ROM is of the mapping format specified in the CANopen specifications and described in Figure 3.

FIGURE 3: MAPPING FORMAT FOR ROM DATA

<table>
<thead>
<tr>
<th>Index</th>
<th>Subindex</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bits</td>
<td>8 bits</td>
<td>8 bits</td>
</tr>
</tbody>
</table>
AN945

EXAMPLE 3: PDO MAPPING DICTIONARY ENTRY

```c
#define DICTIONARY_PDO1_RX_MAP
   \{0x1600,0x00,CONST,1,\{rom unsigned char *)&rMaxIndex2\}\,\\n   \{0x1600,0x01,CONST,4,\{rom unsigned char *)&uRPDO1Map\}\,\\n   \{0x1600,0x02,CONST,4,\{rom unsigned char *)&uPDO1Dummy\}\,\\n   \{0x1600,0x03,CONST,4,\{rom unsigned char *)&uPDO1Dummy\}\,\\n   \{0x1600,0x04,CONST,4,\{rom unsigned char *)&uPDO1Dummy\}\,\\n   \{0x1600,0x05,CONST,4,\{rom unsigned char *)&uPDO1Dummy\}\,\\n   \{0x1600,0x06,CONST,4,\{rom unsigned char *)&uPDO1Dummy\}\,\\n   \{0x1600,0x07,CONST,4,\{rom unsigned char *)&uPDO1Dummy\}\,\\n   \{0x1600,0x08,CONST,4,\{rom unsigned char *)&uPDO1Dummy\}\```

EXAMPLE 4: DICTIONARY STRUCTURE

```c
rom unsigned long uTPDO1Map = 0x60000108;
rom unsigned long uRPDO1Map = 0x62000108;
rom unsigned long uPDO1Dummy = 0x00000008;
```
Synchronization
PDOs can be synchronized by linking their function to the SYNC object. Synchronization depends on the transmission type. The types defined by the specification are listed in Table 5.
Synchronization is simply a matter of using the CO_COMMSyncEvent() function to handle the PDO endpoint. This is discussed in more detail in the section on sync events (page 27).

Timers
The event timer is supported while the inhibit timer is left up to the application designer to provide. This is primarily due to the fine time resolution required (100 µs). If the application requires the event timer, it is possible to handle the CO_PDO1LSTimerEvent() to get 1 ms tick events.

mRPDOOpen
Open the RPDO endpoint where n represents the PDO number. There are only 4 PDOs available. Typically this function would be called within a RPDO communications object write event. Essentially a PDO communications object write event is generated when a node on the network is requesting to start PDO communications.

Syntax
void mRPDOOpen(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.

Return Values
None

Example
(See following page)
Example

// Process access events to the COB ID
void CO_COMM_RPDO1_COBIDAccessEvent(void)
{
    switch (mCO_DictGetCmd())
    {
        case DICT_OBJ_READ: // Read the object
            // Translate MCHP COB to CANopen COB
            mTOOLS_MCHP2CO(mRPDOGetCOB(1));

            // Return the COBID
            *(unsigned long *)(uDict.obj->pReqBuf) = mTOOLS_GetCOBID();
            break;

        case DICT_OBJ_WRITE: // Write the object
            // Translate the COB to MCHP format
            mTOOLS_CO2MCHP(*(unsigned long *)(uDict.obj->pReqBuf));

            // If the request is to stop the PDO
            if (((UNSIGNED32 *)(&mTOOLS_GetCOBID())).PDO_DIS)
            {
                // And if the COB received matches the stored COB and type then close
                if (!(mTOOLS_GetCOBID() ^ mRPDOGetCOB(1)) & 0xFFFFEFFF))
                {
                    // but only close if the PDO endpoint was open
                    if (!mRPDOIsOpen(1)) {mRPDOClose(1);}

                    // Indicate to the local object that this PDO is disabled
                    (*(UNSIGNED32 *)(&mRPDOGetCOB(1))).PDO_DIS = 1;
                }
                else {mCO_DictSetRet(E_PARAM_RANGE);} //error
            }

            // Else if the RPDO is not open then start the RPDO
            else
            {
                // And if the COB received matches the stored COB and type then open
                if (!(mTOOLS_GetCOBID() ^ mRPDOGetCOB(1)) & 0xFFFFEFFF))
                {
                    // but only open if the PDO endpoint was closed
                    if (!mRPDOIsOpen(1)) {mRPDOOpen(1);}

                    // Indicate to the local object that this PDO is enabled
                    (*(UNSIGNED32 *)(&mRPDOGetCOB(1))).PDO_DIS = 0;
                }
                else {mCO_DictSetRet(E_PARAM_RANGE);} //error
            }
            break;
    }
}
mRPDOIsOpen
Query to determine if the RPDO is open. Typically this should be called within a PDO communications object event.

Syntax
BOOL mRPDOIsOpen(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.

Return Values
TRUE: The RPDO is open and accepting messages.
FALSE: The RPDO is closed and will not accept messages.

Example
Refer to the example provided in mRPDOOpen.

mRPDOClose
Close the RPDO endpoint. Typically this should be called within a PDO communications object event.

Syntax
void mRPDOClose(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.

Return Values
None

Example
Refer to the example provided in mRPDOOpen.

mRPDOIsGetRdy
This function queries the Communications Manager for any new received PDOs where n represents the PDO number.

Syntax
BOOL mRPDOIsGetRdy(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.

Return Values
TRUE: Data has been received and is ready to be processed.
FALSE: No data is available yet.

Example
(See following page)
Example

```c
void DemoProcessEvents(void)
{
  unsigned char change;
  unsigned char rise;
  unsigned char fall;

  // Read the input port
  (*(UNSIGNED8 *)uLocalXmtBuffer).bits.b0 = PORTBbits.RB5;
  (*(UNSIGNED8 *)uLocalXmtBuffer).bits.b1 = PORTBbits.RB4;

  // Determine the change if any
  change = uIOinDigiInOld ^ uLocalXmtBuffer[0];
  // Determine if there were any rise events
  rise = (uIOinIntRise & change) & uLocalXmtBuffer[0];
  // Determine if there were any fall events
  fall = (uIOinIntFall & change) & ~uLocalXmtBuffer[0];
  // Determine if there were any change events
  change = (uIOinIntChange & change);
  // Cycle the current value to the old
  uIOinDigiInOld = uLocalXmtBuffer[0];
  // If any of these are true then indicate an interrupt condition
  if (uIOinIntEnable & (change | rise | fall)) uDemoState.bits.b1 = 1;

  if (uDemoState.bits.b1)
  {
    switch (uDemoSyncSet)
    {
      case 0: // Acyclic synchronous transmit
        // Set a synchronous transmit flag
        uDemoState.bits.b2 = 1;
        break;
      case 254: // Asynchronous transmit
      case 255:
        // Reset the asynchronous transmit flag
        uDemoState.bits.b0 = 1;
        break;
    }
  }

  // If ready to send
  if (mTPDOIsPutRdy(1) && uDemoState.bits.b0)
  {
    // Tell the stack that data is loaded for transmit
    mTPDOWritten(1);

    // Reset any synchronous or asynchronous flags
    uDemoState.bits.b0 = 0;
    uDemoState.bits.b1 = 0;
  }

  // If any data has been received
  if (mRPDOIsGetRdy(1))
  {
    // Write out the first byte of the buffer
    LATD = uLocalRcvBuffer[0];

    // PDO read, free the driver to accept more data
    mRPDORead(1);
  }
}
```
mRPDORead
This function is called to indicate to the Communications Manager that the last message it received has been read and processed as necessary. This allows the Communications Manager to accept another PDO message from the driver. The application could simply copy the data or even process the data in-line.

Syntax
void mRPDORead(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.

Return Values
None

Example
Refer to the example provided in mRPDORIsGetRdy().

mRPDOSetCOB
This function sets the RPDO COB ID, where n represents the PDO number (valid range from 1 to 4). This could be set prior to opening the PDO. The COB ID must be in the Microchip standard format.

Syntax
void mRPDOSetCOB(const unsigned char PDOnum, unsigned long rpdoCOB)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.
unsigned long rpdoCOB: The COB ID received by this PDO.

Return Values
None

Example
(See following page)
Example

void DemoInit(void)
{
    // Port D is all output
    LATD = 0;
    TRISD = 0;

    uDemoSyncSet = 255;

    uIoInFilter = 0;
    uIoInPolarity = 0;
    uIoInIntChange = 1;
    uIoInIntRise = 0;
    uIoInIntFall = 0;
    uIoInIntEnable = 1;

    uIoInDigiInOld = uLocalXmtBuffer[0] = 0;
    uLocalRcvBuffer[1] = uLocalXmtBuffer[1] = 0;

    // Convert to MCHP
    mTOOLS_CO2MCHP(mCOMM_GetNodeID().byte + 0xC0000180L);

    // Store the COB
    mTPDOSetCOB(1, mTOOLS_GetCOBID());

    // Convert to MCHP
    mTOOLS_CO2MCHP(mCOMM_GetNodeID().byte + 0xC0000200L);

    // Store the COB
    mRPDOSetCOB(1, mTOOLS_GetCOBID());

    // Set the pointer to the buffers
    mTPDOSetTxPtr(1, (unsigned char *)&uLocalXmtBuffer[0]);

    // Set the pointer to the buffers
    mRPDOSetRxPtr(1, (unsigned char *)&uLocalRcvBuffer[0]);

    // Set the length
    mTPDOSetLen(1, 8);
}
mRPDOGetCOB
This function gets the RPDO COB ID currently used.

Syntax
unsigned long mRPDOGetCOB(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.

Return Values
unsigned long: The COB ID received by this PDO.

Example
Refer to the example provided in mRPDOOpen.

mRPDOGetLen
This function gets the length of the last received PDO.

Syntax
unsigned char mRPDOGetLen(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.

Return Values
unsigned char: The length of the message, valid values from 0 to 8 bytes.

Example
None

mRPDOGetRxPtr
This function gets the stored pointer to the local receive buffer. The pointer must be set prior to opening communications to the endpoint. When communications is open all messages will be stored in the location referenced by this pointer.

Syntax
unsigned char * mRPDOGetRxPtr(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.
unsigned char * pRXBUF

Return Values
unsigned char*: Pointer to the buffer space

Example
None
mRPDAOSetRxPtr
This function sets the pointer to the local receive buffer. The pointer must be set prior to opening communications to the endpoint. When communications are open all messages will be stored in the location referenced by this pointer.

Syntax
void mRPDAOSetRxPtr(const unsigned char PDOnum, unsigned char *pRXBUF)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.
unsigned char *pRXBUF

Return Values
None

Example
Refer to the example provided in mRPDAOSetCOB().

mTPDOOpen
Open the TPDO endpoint. There are only four PDOs available. Typically this should be called within a TPDO communications object write event. Essentially a PDO communications object write event is generated when a node on the network is requesting to start PDO communications.

Syntax
void mTPDOOpen(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.

Return Values
None

Example
(See following page)
Example

// Process access events to the COB ID
void CO_COMM_TPDO1_COBIDAccessEvent(void)
{
    switch (mCO_DictGetCmd())
    {
    case DICT_OBJ_READ: // Read the object
        // Translate MCHP COB to CANopen COB
        mTOOLS_MCHP2CO(mTPDOGetCOB(1));

        // Return the COBID
        *(unsigned long *)(uDict.obj->pReqBuf) = mTOOLS_GetCOBID();
        break;

    case DICT_OBJ_WRITE: // Write the object
        // Translate the COB to MCHP format
        mTOOLS_CO2MCHP(*(unsigned long *)(uDict.obj->pReqBuf));

        // If the request is to stop the PDO
        if (((UNSIGNED32 *)&mTOOLS_GetCOBID()).PDO_DIS)
        {
            // And if the COB received matches the stored COB and type then close
            if (!((mTOOLS_GetCOBID() ^ mTPDOGetCOB(1)) & 0xFFFFEFFF))
            {
                // but only close if the PDO endpoint was open
                if (!mTPDOIsOpen(1)) {mTPDOClose(1);}

                // Indicate to the local object that this PDO is disabled
                (*(UNSIGNED32 *)&mTPDOGetCOB(1)).PDO_DIS = 1;
            }
            else {mCO_DictSetRet(E_PARAM_RANGE);} //error
        }

        // Else if the TPDO is not open then start the TPDO
        else
        {
            // And if the COB received matches the stored COB and type then open
            if (!((mTOOLS_GetCOBID() ^ mTPDOGetCOB(1)) & 0xFFFFEFFF))
            {
                // but only open if the PDO endpoint was closed
                if (!mTPDOIsOpen(1)) {mTPDOOpen(1);}

                // Indicate to the local object that this PDO is enabled
                (*(UNSIGNED32 *)&mTPDOGetCOB(1)).PDO_DIS = 0;
            }
            else {mCO_DictSetRet(E_PARAM_RANGE);} //error
        }
    }
}
mTPDOIsOpen
Query to determine if the TPDO is open. Typically this should be called within a PDO communications object event.

Syntax
BOOL mTPDOIsOpen(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.

Return Values
TRUE: The Communications Manager is ready to accept new data.
FALSE: The Communications Manager is busy transmitting the previous message.

Example
Refer to the example provided in mTPDOOpen().

mTPDOClose
Close the TPDO endpoint where n represents the PDO number (valid range from 1 to 4). Typically this should be called within a PDO communications object event.

Syntax
void mTPDOClose(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.

Return Values
None

Example
Refer to the example provided in mTPDOOpen().

mTPDOIsPutRdy
This function queries the Communications Manager for an available slot for transmitting a PDO. This function will return true if the manager is ready to accept a message to send on the bus.

Syntax
BOOL mTPDOIsPutRdy(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.

Return Values
TRUE: The Communications Manager is ready to accept new data.
FALSE: The Communications Manager is busy transmitting the previous message.

Example
Refer to the example provided in mRPDOIsGetRdy().
mTPDOWritten
Indicates to the Communications Manager that a message has been loaded for the manager to send. This allows the Communications Manager to queue the message for transmission. The CO_PDOTXFinEvent() event function is called when the message is placed on the bus.

Syntax
void mTPDOWritten(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.

Return Values
None

Example
Refer to the example provided in mRPDOIsGetRdy().

mTPDOSetCOB
This function sets the TPDO COB ID. This should be set prior to sending a TPDO. The COB ID must be in the Microchip standard format.

Syntax
void mTPDOSetCOB(const unsigned char PDOnum, unsigned long tpdoCOB)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.
unsigned long tpdoCOB: The COB ID to be sent.

Return Values
None

Example
Refer to the example provided in mRPDOSetCOB().

mTPDOGetCOB
This function gets the TPDO COB ID currently used.

Syntax
unsigned long mTPDOGetCOB(const unsigned char PDOnum)

Parameters
const unsigned char PDOnum: Valid range of 1 to 4. Must be an actual number, not a macro.

Return Values
unsigned long: The COB ID currently used by this PDO.

Example
Refer to the example provided in mRPDOSetCOB().
**mTPDOSetLen**

This function sets the TPDO data length. The length must be between 0 and 8.

**Syntax**

```
unsigned long mTPDOSetLen(const unsigned char PDOnum, unsigned char length)
```

**Parameters**

- `const unsigned char PDOnum`: Valid range of 1 to 4. Must be an actual number, not a macro.
- `unsigned char length`: The length of the PDO, must be from 0 to 8 bytes.

**Return Values**

None

**Example**

Refer to the example provided in `mRPDOSetCOB()`.

**mTPDOGetTxPtr**

This function gets the pointer currently pointing to the local transmit buffer. When transmitting, all messages will be transmitted from the location referenced by this pointer.

**Syntax**

```
unsigned char * mTPDOGetTxPtr(const unsigned char PDOnum)
```

**Parameters**

- `const unsigned char PDOnum`: Valid range of 1 to 4. Must be an actual number, not a macro.

**Return Values**

`unsigned char *`: Returns the currently used pointer to the buffer

**Example**

None

**mTPDOSetTxPtr**

This function sets the pointer to the local transmit buffer. When transmitting, all messages will be transmitted from the location referenced by this pointer.

**Syntax**

```
void mTPDOSetTxPtr(const unsigned char PDOnum)
```

**Parameters**

- `const unsigned char PDOnum`: Valid range of 1 to 4. Must be an actual number, not a macro.

**Return Values**

None

**Example**

Refer to the example provided in `mRPDOIsGetRdy()`.
CO_PDOnLSTimerEvent
This is the timer event callback function. This function is called every 1 ms if the PDO is enabled. Typically the application could use this for the PDO event timer function specified in CANopen.

Syntax
void CO_PDOnLSTimerEvent(void)

Parameters
None

Return Values
None

Example
None

CO_PDOnTXFinEvent
This is the transmit finished event callback function. This event is generated when a message that was queued to transmit has been placed on the CAN.

Syntax
void CO_PDOnTxFinEvent(void)

Parameters
None

Return Values
None

Example
None
SYNC Events and Services

There is only one event that is received from theSYNC object; it is the CO_COMMSyncEvent(). This event is generated only when a SYNC message is received, and it is used for synchronized PDO processing. This event should be handled in the application’s PDO message processing.

CO_COMMSyncEvent

This is the only event that is generated from the SYNC object. This event is generated only when a SYNC message is received, and it is used for synchronized PDO processing. This event should be handled in the application’s PDO message processing.

Syntax

void CO_COMMSyncEvent(void)

Parameters

None

Return Values

None

Example

This is a simple example of a handling function for a variable synchronous PDO Type that is cyclic in nature. This is defined by a PDO Type (TPDO communications parameter at subindex 2) that is between 1 and 240 inclusive.

void CO_COMMSyncEvent(void)
{
    // Process only if in a synchronous mode
    if ((uDemoSyncSet == 0) && (uDemoState.bits.b2))
    {
        // Reset the synchronous transmit and transfer to async
        uDemoState.bits.b2 = 0;
        uDemoState.bits.b0 = 1;
    }
    else
    if ((uDemoSyncSet >= 1) && (uDemoSyncSet <= 240))
    {
        // Adjust the sync counter
        uDemoSyncCount--;

        // If time to generate sync
        if (uDemoSyncCount == 0)
        {
            // Reset the sync counter
            uDemoSyncCount = uDemoSyncSet;

            // Start the PDO transmission
            uDemoState.bits.b0 = 1;
        }
    }
}
mSYNC_SetCOBID
This function is used to set the COB ID for the SYNC object. This should be called at least once before initializing to properly set the COB ID within the firmware.

Syntax
void mSYNC_SetCOBID(unsigned long SYNC_COB)

Parameters
The COB ID in the Microchip format.
unsigned long SYNC_COB

Return Values
None

Example
Refer to the example provided in mCO_ProcessAllEvents.

mSYNC_GetCOBID
This function is used to get the COB ID currently used for the SYNC object.

Syntax
unsigned long mSYNC_GetCOBID(void)

Parameters
None

Return Values
unsigned long SYNC_COB: The COB ID in the Microchip format.

Example
None
Network Management Events and Services

Network management is provided through the NMT object, which essentially encompasses the node state machine (see Figure 2).

There are a handful of services provided to enter the node into a particular state. However, the state will change through normal network management requests from the NMT master. When a state is changed due to a request from the master, then an event is generated. All the events and services are listed below.

**mNMT_Start**

Call this function to start communications that have been stopped. Typically this is automatically called by the NMT managing routines as a result of a NMT request from the master to set the appropriate state.

**Syntax**

```c
void mNMT_Start(void)
```

**Parameters**

None

**Return Values**

None

**Example**

None

**mNMT_Stop**

Call this function to stop a node that was in the operational or preoperational state. Typically this is automatically called by the NMT managing routines as a result of a NMT request from the master to set the appropriate state.

**Syntax**

```c
void mNMT_Stop(void)
```

**Parameters**

None

**Return Values**

None

**Example**

None
**mNMT_GotoPreopState**

Call this function to place the node into the preoperational state. Typically this is automatically called by the NMT managing routines as a result of an NMT request from the master to set the appropriate state.

**Syntax**

```
void mNMT_GotoPreopState(void)
```

**Parameters**

None

**Return Values**

None

**Example**

None

---

**mNMT_GotoOperState**

Call this function to place the node into the operational state. Typically this is automatically called by the NMT managing routines as a result of an NMT request from the master to set the appropriate state.

**Syntax**

```
void mNMT_GotoOperState(void)
```

**Parameters**

None

**Return Values**

None

**Example**

None

---

**mNMT_StateIsStopped**

Query to determine if the node is currently in a stopped state.

**Syntax**

```
BOOL mNMT_StateIsStopped(void)
```

**Parameters**

None

**Return Values**

- **TRUE:** If node is in STOPPED state.
- **FALSE:** If node is in PREOPERATIONAL or OPERATIONAL state.

**Example**

None
mNMT_StateIsOperational
Query to determine if the node is currently in the operational state.

Syntax
BOOL mNMT_StateIsOperational(void)

Parameters
None

Return Values
TRUE: If node is in OPERATIONAL state.
FALSE: If node is STOPPED or PREOPERATIONAL state.

Example
None

mNMT_StateIsPreOperational
Query to determine if the node is currently in the operational state.

Syntax
BOOL mNMT_StateIsPreOperational(void)

Parameters
None

Return Values
TRUE: If node is in PREOPERATIONAL state.
FALSE: If node is in STOPPED or OPERATIONAL state.

Example
None

CO_NMTStateChangeEvent
This callback function is called when the state of the system has been changed through Network Management Request.

Syntax
void CO_NMTStateChangeEvent(void)

Parameters
None

Return Values
None

Example
None
CO_NMTResetEvent
This callback function is called when a communications Reset has been requested. The communications is automatically reset after this event is handled.

Syntax
void CO_NMTStateChangeEvent(void)

Parameters
None

Return Values
None

Example
None

CO_NMTAppResetRequest
This callback function is called when an application Reset has been requested. How this event is handled depends on the application design. After handling this event the CO_COMMResetEvent() event will be generated. The communications are automatically reset after the CO_COMMResetEvent() event is handled.

Syntax
void CO_NMTAppResetRequest(void)

Parameters
None

Return Values
None

Example
None
Node Guard/Heartbeat Events and Services

A combined Node Guard/Heartbeat object is provided as required by the specification. There are a small number of services provided to initialize and get information about the object.

There is only one possible event generated by the Node Guard/Heartbeat object, which relates specifically to the node guard half of the object. The CO_NMTENodeGuardErrEvent() function is called when the lifetime of the object has been exceeded. The lifetime is defined in the specification as the product of the lifetime factor and the guard time.

mNMTE_SetHeartBeat

Call this function to set the Heartbeat. The Heartbeat is an unsigned long in the format specified by the CANopen specifications. This should be set prior to initializing communications.

Syntax

void mNMTE_SetHeartBeat(unsigned long HeartBeat)

Parameters

unsigned long HeartBeat

Return Values

None

Example

None

mNMTE_GetHeartBeat

Use this function to return the current Heartbeat setting. An unsigned long is returned.

Syntax

unsigned long mNMTE_GetHeartBeat(void)

Parameters

None

Return Values

unsigned long HeartBeat

Example

None
mNMTE_SetGuardTime
Call this function to set the guard time. The guard time is an unsigned long in the format specified by the CANopen specifications. This should be set prior to initializing communications.

Syntax
void mNMTE_SetGuardTime(unsigned long GuardTime)

Parameters
None

Return Values
None

Example
None

mNMTE_GetGuardTime
Use this function to return the current guard time setting. An unsigned long is returned.

Syntax
unsigned long mNMTE_GetGuardTime(void)

Parameters
None

Return Values
unsigned long GuardTime

Example
None

mNMTE_SetLifeFactor
Use this function to set the current guard time setting. An unsigned long is returned.

Syntax
void mNMTE_SetLifeFactor(unsigned char LifeFactor)

Parameters
None

Return Values
None

Example
None
mNMTE_GetLifeFactor

Use this function to return the current guard time setting. An unsigned char long is returned.

Syntax

unsigned char mNMTE_GetLifeFactor(void)

Parameters

None

Return Values

unsigned char LifeFactor

Example

None

CO_NMTENodeGuardErrEvent

This callback function is called when there is a node guard event. A node guard event occurs when a node guard message is not received within the defined lifetime (the product of life time factor and guard time). How this event is handled is dependent on the application.

Syntax

void CO_NMTENodeGuardErrEvent(void)

Parameters

None

Return Values

None

Example

None
Objects and the Object Dictionary

In this design each dictionary entry is a structure within program memory. Within each structure is the necessary information to identify the object and its location. The identity is flexible enough that more than simple data types, arrays, and structures can be defined as objects. A function can be defined as an object as well, and this is where the true flexibility lies for complex objects.

THE OBJECT STRUCTURE

An object defined in the Object Dictionary is stored in program memory; its structure is shown in Example 5. This structure contains enough information to describe any object.

- index: the index of the object
- subindex: the subindex of the object
- ctl: the control byte. This defines the type of object.
- len: the length of the object in bytes.
- *pROM: a pointer to the object or object handling function. The pointer should always be cast to rom unsigned char *.

EXAMPLE 5: DICTIONARY STRUCTURE

```c
typedef struct _DICTIONARY_OBJECT_TEMPLATE {
    unsigned int index;
    unsigned char subindex;
    unsigned char ctl;
    unsigned int len;
    rom unsigned char * pROM;
} DICT_OBJECT_TEMPLATE;
```

OBJECT GROUPS

The Object Dictionary is broken into groups for faster dictionary searching. Thus every entry within the Object Dictionary must be stored within the appropriate group. Table 6 identifies all the groups. Any entries in the dictionary should be placed in numerical order within the appropriate group.

OBJECT CONTROL BITS

How an object is handled within the dictionary depends on its control bits. An object could be read/write, read only, or even functionally defined to accommodate very unique objects. Table 7 defines the bits of the object control byte.

To easily manipulate individual bits within the control byte, a series of symbolic bit modifiers have been provided. Table 8 provides the logical AND modifiers to control the object. These can be combined manually to form a specific control. For example, the following statement defines an object that is readable, writable, defined as a function, and mappable:

```
RD & WR & N_ROM & N_EE & FDEF & MAP & N_FSUB
```

In a similar fashion, Table 9 provides the typical logical OR modifier definitions to control the object. These can also be combined with the bit names shown in Table 8. For example, the following statement defines an object that is readable, writable, defined as a function, and mappable (same as previous):

```
RW | FUNC | MAP_BIT
```

Several examples of the usage of bit modifiers are shown in Example 6, in entries 4, 8, 9 and 10.
### TABLE 6: OBJECT GROUPS

<table>
<thead>
<tr>
<th>Object Group Name</th>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DICTIONARY_DATA_TYPES</td>
<td>0000h</td>
<td>Data types defined in the object dictionary. Although data types are defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>within the object dictionary, the specification indicates that support is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not required.</td>
</tr>
<tr>
<td>DICTIONARY_DEVICE_INFO</td>
<td>1000h</td>
<td>This group is within the CANopen communications section and contains the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>device specific information including COBIDs, certain endpoints, and status.</td>
</tr>
<tr>
<td>DICTIONARY_SDO</td>
<td>1200h</td>
<td>One group for SDO parameters is provided.</td>
</tr>
<tr>
<td>DICTIONARY_PDO1_RX_COMM</td>
<td>1400h</td>
<td>Individual groups are provided for four RPDO communications parameters.</td>
</tr>
<tr>
<td>DICTIONARY_PDO2_RX_COMM</td>
<td>1401h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_PDO3_RX_COMM</td>
<td>1402h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_PDO4_RX_COMM</td>
<td>1403h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_PDO1_RX_MAP</td>
<td>1600h</td>
<td>Individual groups are provided for four RPDO mapping parameters.</td>
</tr>
<tr>
<td>DICTIONARY_PDO2_RX_MAP</td>
<td>1601h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_PDO3_RX_MAP</td>
<td>1602h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_PDO4_RX_MAP</td>
<td>1603h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_PDO1_TX_COMM</td>
<td>1800h</td>
<td>Individual groups are provided for four TPDO communications parameters.</td>
</tr>
<tr>
<td>DICTIONARY_PDO2_TX_COMM</td>
<td>1801h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_PDO3_TX_COMM</td>
<td>1802h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_PDO4_TX_COMM</td>
<td>1803h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_PDO1_TX_MAP</td>
<td>1A00h</td>
<td>Individual groups are provided for four TPDO mapping parameters.</td>
</tr>
<tr>
<td>DICTIONARY_PDO2_TX_MAP</td>
<td>1A01h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_PDO3_TX_MAP</td>
<td>1A02h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_PDO4_TX_MAP</td>
<td>1A03h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_MANUFACTURER_SPECIFIC_1</td>
<td>2000h</td>
<td>These groups are provided for manufacturer specific objects.</td>
</tr>
<tr>
<td>DICTIONARY_MANUFACTURER_SPECIFIC_2</td>
<td>3000h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_MANUFACTURER_SPECIFIC_3</td>
<td>4000h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_MANUFACTURER_SPECIFIC_4</td>
<td>5000h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_STANDARD_1</td>
<td>6000h</td>
<td>These groups are provided for CANopen standard objects.</td>
</tr>
<tr>
<td>DICTIONARY_STANDARD_2</td>
<td>7000h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_STANDARD_3</td>
<td>8000h</td>
<td></td>
</tr>
<tr>
<td>DICTIONARY_STANDARD_4</td>
<td>9000h</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 7: CONTROL BIT DEFINITIONS

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0</td>
<td>RD_BIT</td>
<td>This bit defines the read access of the object. If this bit is set then the object is readable from a node on the network.</td>
</tr>
<tr>
<td>Bit 1</td>
<td>WR_BIT</td>
<td>This bit defines the write access of the object. If this bit is set then the object is writable by a node on the network.</td>
</tr>
<tr>
<td>Bit 2</td>
<td>ROM_BIT</td>
<td>This bit defines an object that is located within ROM. Setting this bit does not imply the object cannot be written. This only defines the location where this bit is stored.</td>
</tr>
<tr>
<td>Bit 3</td>
<td>EE_BIT</td>
<td>This bit defines an object that is located in EEPROM. Note, no automatic handling is provided at this time for EEPROM. If the EE_BIT is set then the FDEF_BIT should also be set so the dictionary access tools know that the application designer is handling access to EEDATA memory through a custom function.</td>
</tr>
<tr>
<td>Bit 4</td>
<td>FDEF_BIT</td>
<td>This bit defines an object that is functionally defined. Typically objects are defined by a function if they have special rules that cannot be defined by a single static type. For example, an object that triggers an event when read should be functionally defined. Or if an object can change read-write access level based on application dependent events or states should also be functionally defined. Also note, if this bit is set then all other bits can be defined within the object handling function, except the FSUB_BIT.</td>
</tr>
<tr>
<td>Bit 5</td>
<td>MAP_BIT</td>
<td>This bit defines the mappability of the object. Thus if this bit is set then the object can be mapped into a PDO.</td>
</tr>
<tr>
<td>Bit 6</td>
<td>FSUB_BIT</td>
<td>This bit defines whether the entire subindex array is functionally defined. Thus for a particular index there will be only one entry in the dictionary. And all requests to access any subindex are handled by the object's access handling function. This is useful for objects where all of the subindices have the same functionality but require different parameter values; therefore, only one entry is required in the dictionary file.</td>
</tr>
<tr>
<td>Bit 7</td>
<td>reserved</td>
<td>reserved at this time</td>
</tr>
</tbody>
</table>

### TABLE 8: LOGIC AND BIT DEFINITIONS

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RD</td>
<td>Allow read</td>
</tr>
<tr>
<td>N_RD</td>
<td>Read not allowed</td>
</tr>
<tr>
<td>WR</td>
<td>Write allowed</td>
</tr>
<tr>
<td>N_WR</td>
<td>Write not allowed</td>
</tr>
<tr>
<td>ROM</td>
<td>ROM based object</td>
</tr>
<tr>
<td>N_ROM</td>
<td>Not a ROM based object</td>
</tr>
<tr>
<td>EE</td>
<td>EEDATA based object</td>
</tr>
<tr>
<td>N_EE</td>
<td>Not an EEDATA based object</td>
</tr>
<tr>
<td>FDEF</td>
<td>Functionally defined object</td>
</tr>
<tr>
<td>N_FDEF</td>
<td>Not a functionally defined object</td>
</tr>
<tr>
<td>MAP</td>
<td>Mappable object</td>
</tr>
<tr>
<td>N_MAP</td>
<td>Not a mappable object</td>
</tr>
<tr>
<td>FSUB</td>
<td>Functionally defined subindex</td>
</tr>
<tr>
<td>N_FSUB</td>
<td>Not a functionally defined subindex</td>
</tr>
</tbody>
</table>

### TABLE 9: LOGIC OR BIT DEFINITIONS

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONST</td>
<td>ROM based read-only object</td>
</tr>
<tr>
<td>RW</td>
<td>Readable and writable object</td>
</tr>
<tr>
<td>RO</td>
<td>Read-only object</td>
</tr>
<tr>
<td>WO</td>
<td>Write-only object</td>
</tr>
<tr>
<td>RW_EE</td>
<td>Readable and writable EEDATA object</td>
</tr>
<tr>
<td>RO_EE</td>
<td>Read-only object in EEDATA</td>
</tr>
<tr>
<td>WO_EE</td>
<td>Write-only object in EEDATA</td>
</tr>
<tr>
<td>FUNC</td>
<td>Functionally defined object</td>
</tr>
</tbody>
</table>
SIMPLE OBJECTS

The dictionary provides support for simple objects. Simple objects are essentially objects that operate within the realm of a normal data type. This includes any data type supported by the compiler as well as arrays.

A simple object is defined in the object dictionary by referencing the object within the dictionary. This is illustrated by the first dictionary entry in Example 7. A read request to this object will return the data stored in uCO_DevManufacturerStatReg; a write request will return an error, since this is a read-only object.

FUNCTIONALLY DEFINED OBJECT

Objects are defined by a function when the object has some properties that do not follow a standard data type or array defined in the C language. For example, a variable unsigned char MyObj that has no unusual conditions does not need to be defined by a function; however, if in MyObj bit 7 enables the write to MyObj, then this would require special handling and must be defined by a function, similar to COB IDs.

An object is defined by a function when the FDEF_BIT is set in its control byte. This is demonstrated with the second dictionary entry in Example 7, which defines the COB ID for the SYNC object. In this case, the function _CO_COMM_SYNC_COBIDAccessEvent() is called when there is a request to access the object at index 1005h, subindex 0x00.

WRITING AN OBJECT HANDLING FUNCTION

An object is referenced through an SDO, PDO, or through some application access. If the object is defined by a function then the function defined in the dictionary will be called when the object is referenced. There are three possible events that the object handling function can handle when referenced:

- Read control: Read the control bits defined by the function. This applies to all bits except the FSUB_BIT and FDEF_BIT bits; these bits must be defined for the object within the dictionary.
- Read: Read the object if it is readable.
- Write: Write the object if it is writable.

Example 8 demonstrates what a typical handling function looks like. Example 9 is an example of a handler for the TPDO1 COB ID object.

An object handling function is provided with functions and a structure to process requests to or from. The functions are mCO_DictGetCmd() and mCO_DictSetRet(). The first is used to retrieve the command, and the second is used to return any errors to the requestor. Table 11 lists the errors that can be returned. In the case of a successful request, then no response is necessary; the dictionary assumes success.

The requestor will set a pointer in the dictionary (uDict.obj) to its local DICT_OBJ structure. This structure contains information about the object as well as the requestor. The structure is defined in Table 8.

EXAMPLE 7: EXAMPLES OF OBJECT DEFINITIONS

```
Simple Object Definition:
{0x1002,0x00,RO,4,{(rom unsigned char *)&uCO_DevManufacturerStatReg}}

Functionally Defined Object:
{0x1005,0x00,FUNC | RW,4,{(rom unsigned char *)&_CO_COMM_SYNC_COBIDAccessEvent}
```
### TABLE 10: DICT_OBJ UDICT STRUCTURE

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pReqBuf</td>
<td>unsigned char *</td>
<td>Pointer to the requestor’s buffer. This is the pointer to the requestor’s data when writing an object. When reading, this is the pointer to the requestor’s buffer space.</td>
</tr>
<tr>
<td>reqLen</td>
<td>unsigned int</td>
<td>Number of bytes requested. This should never exceed the length of the object.</td>
</tr>
<tr>
<td>reqOffst</td>
<td>unsigned int</td>
<td>Starting point for the request. This is provided to support partial requests due to low buffer space. This is most useful for read requests; for write requests this would be unlikely since partially writing an object is not always desirable. Also, this parameter does not need to be supported if the number of bytes in the object is less than 8.</td>
</tr>
<tr>
<td>index</td>
<td>unsigned int</td>
<td>CANopen Index.</td>
</tr>
<tr>
<td>subindex</td>
<td>unsigned char</td>
<td>CANopen subindex.</td>
</tr>
<tr>
<td>ctl</td>
<td>enum DICT_CTL</td>
<td>Memory access type.</td>
</tr>
<tr>
<td>len</td>
<td>unsigned int</td>
<td>Size of the object in bytes.</td>
</tr>
<tr>
<td>p</td>
<td>union DICT_PTRS</td>
<td>Pointers to objects.</td>
</tr>
</tbody>
</table>

### TABLE 11: ERROR DEFINITIONS

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_SUCCESS</td>
<td>Success, no error</td>
</tr>
<tr>
<td>E_TOGGLE</td>
<td>Toggle bit not alternated</td>
</tr>
<tr>
<td>E_SDO_TIME</td>
<td>SDO protocol timed out</td>
</tr>
<tr>
<td>E_CS_CMD</td>
<td>Client/server command specifier not valid or unknown</td>
</tr>
<tr>
<td>E_MEMORY_OUT</td>
<td>Out of memory</td>
</tr>
<tr>
<td>E_UNSUPP_ACCESS</td>
<td>Unsupported access to object</td>
</tr>
<tr>
<td>E_CANNOT_READ</td>
<td>Attempt to read a write only object</td>
</tr>
<tr>
<td>E_CANNOT_WRITE</td>
<td>Attempt to write a read-only object</td>
</tr>
<tr>
<td>E_OBJ_NOT_FOUND</td>
<td>Object does not exist in the object dictionary</td>
</tr>
<tr>
<td>E_OBJ_CANNOT_MAP</td>
<td>Object cannot be mapped to the PDO</td>
</tr>
<tr>
<td>E_OBJ_MAP_LEN</td>
<td>The number and length of the objects to be mapped would exceed PDO length</td>
</tr>
<tr>
<td>E_GEN_PARAM_COMP</td>
<td>General parameter incompatibility</td>
</tr>
<tr>
<td>E_GEN_INTERNAL_COMP</td>
<td>General internal incompatibility in the device</td>
</tr>
<tr>
<td>E_HARDWARE</td>
<td>Access failure due to a hardware error</td>
</tr>
<tr>
<td>E_LEN_SERVICE</td>
<td>Data type does not match, length of service parameter does not match</td>
</tr>
<tr>
<td>E_LEN_SERVICE_HIGH</td>
<td>Data type does not match, length of service parameter too high</td>
</tr>
<tr>
<td>E_LEN_SERVICE_LOW</td>
<td>Data type does not match, length of service parameter too low</td>
</tr>
<tr>
<td>E_SUBINDEX_NOT_FOUND</td>
<td>Subindex does not exist</td>
</tr>
<tr>
<td>E_PARAM_RANGE</td>
<td>Value range of parameter exceeded (only for write access)</td>
</tr>
<tr>
<td>E_PARAM_HIGH</td>
<td>Value of parameter too high</td>
</tr>
<tr>
<td>E_PARAM_LOW</td>
<td>Value of parameter too low</td>
</tr>
<tr>
<td>E_MAX_LT_MIN</td>
<td>Maximum value is less than minimum value</td>
</tr>
<tr>
<td>E_GENERAL</td>
<td>General error</td>
</tr>
<tr>
<td>E_TRANSFER</td>
<td>Data cannot be transferred or stored to the application</td>
</tr>
<tr>
<td>E_LOCAL_CONTROL</td>
<td>Data cannot be transferred or stored to the application because of local control</td>
</tr>
<tr>
<td>E_DEV_STATE</td>
<td>Data cannot be transferred or stored to the application because of the present device state</td>
</tr>
</tbody>
</table>
EXAMPLE 8: FUNCTIONAL OBJECT HANDLING

```c
void MyObjectHandlingFunction(void)
{
    switch (mCO_DictGetCmd())
    {
    case DICT_OBJ_INFO:// Get information about the object
        // Code in this request type should modify the type of access. For
        // example, if the object can change from RO to RW based on a particular
        // state of the application then this would be handled here. In most
        // situations this can be omitted since the object info is static;
        // static information is supported directly by the dictionary.
        break;
    case DICT_OBJ_READ: // Read the object
        // This is the object read request. Code in this request type should
        // handle any data movement and/or events based on the Read.
        break;
    case DICT_OBJ_WRITE: // Write the object
        // This is the object write request. Code in this request type should
        // handle any data movement and/or events based on the Write.
        break;
    }
}
```
EXAMPLE 9: FUNCTIONAL OBJECT HANDLING EXAMPLE

```c
void CO_COMM_TPDO1_COBIDAccessEvent(void)
{
    switch (mCO_DictGetCmd())
    {
        case DICT_OBJ_READ: // Read the object
            // Translate MCHP COB to CANopen COB
            mTOOLS_MCHP2CO(mTPDOGetCOB(1));

            // Return the COBID
            *(unsigned long *)(uDict.obj->pReqBuf) = mTOOLS_GetCOBID();
            break;

        case DICT_OBJ_WRITE: // Write the object
            // Translate the COB to MCHP format
            mTOOLS_CO2MCHP(*(unsigned long *)(uDict.obj->pReqBuf));

            // If the request is to stop the PDO
            if (/*(UNSIGNED32 *)(mTOOLS_GetCOBID()).PDO_DIS*/) {
                // And if the COB received matches the stored COB and type then close
                if (((mTOOLS_GetCOBID() ^ mTPDOGetCOB(1)) & 0xFFFFEFFF)) {
                    // but only close if the PDO endpoint was open
                    if (mTPDOIsOpen(1)) {mTPDOClose(1);}

                    // Indicate to the local object that this PDO is disabled
                    (*(UNSIGNED32 *)(mTPDOGetCOB(1))).PDO_DIS = 1;
                } else {mCO_DictSetRet(E_PARAM_RANGE);} //error
            }

            // Else if the TPDO is not open then start the TPDO
            else {
                // And if the COB received matches the stored COB and type then open
                if (((mTOOLS_GetCOBID() ^ mTPDOGetCOB(1)) & 0xFFFFEFFF)) {
                    // but only open if the PDO endpoint was closed
                    if (!mTPDOIsOpen(1)) {mTPDOOpen(1);}

                    // Indicate to the local object that this PDO is enabled
                    (*(UNSIGNED32 *)(mTPDOGetCOB(1))).PDO_DIS = 0;
                } else {mCO_DictSetRet(E_PARAM_RANGE);} //error
            }
            break;
    }
}
```
DICTIONARY SERVICES

There are several services for dictionary management available for use by the SDO endpoint. If necessary, they may also be used for dynamic PDO mapping.

mCO_DictObjectRead

This function reads the object defined by myObj. To use this, the object information must be stored locally as a DICT_OBJ structure then passed to the mCO_DictObjectRead() function. Internally only the reference is used.

Within the DICT_OBJ structure is the information necessary for receiving data from the object. Some of this information must be provided by the calling function and other information must be provided by the dictionary. The mCO_DictObjectDecode() function must be called prior to calling mCO_DictObjectRead() to get the access and reference information stored in the dictionary. Other information must be provided by the user. The following table describes the structure and the source of information for each element.

**TABLE 12: DICT_OBJ STRUCTURE**

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Provided by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pReqBuf</td>
<td>unsigned char *</td>
<td>User</td>
<td>Pointer to the requestors buffer</td>
</tr>
<tr>
<td>reqLen</td>
<td>unsigned int</td>
<td>User</td>
<td>Number of bytes requested</td>
</tr>
<tr>
<td>reqOffst</td>
<td>unsigned int</td>
<td>User</td>
<td>Starting point for the request</td>
</tr>
<tr>
<td>index</td>
<td>unsigned int</td>
<td>User</td>
<td>CANopen Index</td>
</tr>
<tr>
<td>subindex</td>
<td>unsigned char</td>
<td>User</td>
<td>CANopen subindex</td>
</tr>
<tr>
<td>ctl</td>
<td>enum DICT_CTL</td>
<td>mCO_DictObjectDecode()</td>
<td>Memory access type</td>
</tr>
<tr>
<td>len</td>
<td>unsigned int</td>
<td>mCO_DictObjectDecode()</td>
<td>Size of the object in bytes</td>
</tr>
<tr>
<td>p</td>
<td>union DICT_PTRS</td>
<td>mCO_DictObjectDecode()</td>
<td>Pointers to objects</td>
</tr>
</tbody>
</table>

**Syntax**

```c
void mCO_DictObjectRead(DICT_OBJ myObj)
```

**Parameters**

DICT_OBJ myObj

**Return Values**

None. Use mCO_DictGetRet() to retrieve the error code.

**Example**

```c
void MyFunc(void)
{
    DICT_OBJ myLocalObj;
    unsigned char localArray[20];
    // Specify the object
    myLocalObj.index = 0x1008L;
    myLocalObj.subindex = 0x00;
    // Get the information stored in the dictionary
    mCO_DictObjectDecode(myLocalObj);
    // Specify the local space and what data to read
    myLocalObj.pReqBuf = localArray;
    myLocalObj.reqLen = 0x8;
    myLocalObj.reqOffst = 0x0;
    // Read the object
    mCO_DictObjectRead(myLocalObj);
}
```
mCO_DictObjectWrite
This function writes the object defined by myObj. To use this, the object information must be stored locally as a DICT_OBJ structure then passed to the mCO_DictObjectWrite() function. Internally only the reference is used.

Syntax
void mCO_DictObjectWrite(DICT_OBJ myObj)

Parameters
DICT_OBJ myObj : The object structure shown in Table 12.

Return Values
None. Use mCO_DictGetRet() to retrieve the error code.

Example
The basic usage is similar to the example given for mCO_DictObjectRead() (page 43).

mCO_DictObjectDecode
This function is used to fill in any static information for a particular object that resides within the dictionary. An object defined by myObj must be declared locally and passed to the function. The function will take the index and sub index information and search for it within the dictionary. If the object is found then a pointer, length, and some control information will be loaded within the myObj structure; refer to Table 12. Status information is returned and can be retrieved with the mCO_DictGetRet() function.

Syntax
void mCO_DictObjectDecode(DICT_OBJ myObj)

Parameters
DICT_OBJ myObj : The object structure shown in Table 12.

Return Values
None. Use mCO_DictGetRet() to retrieve the error code.

Example
The basic usage is similar to the example given for mCO_DictObjectRead() (page 43).
mCO_DictGetCmd
This function is used to retrieve the command for an object. There are only three commands: DICT_OBJ_INFO, DICT_OBJ_READ, and DICT_OBJ_WRITE.

Syntax
enum _DICT_OBJECT_REQUEST mCO_DictGetCmd(void)

Parameters
None

Return Values
DICT_OBJ_INFO: Read object control information.
DICT_OBJ_READ: Read the object.
DICT_OBJ_WRITE: Write the object.

Example
Refer to the code in Example 9 (page 42).

mCO_DictGetRet
This function is used to get the return status of a dictionary operation.

Syntax
unsigned char mCO_DictGetRet(void)

Parameters
None

Return Values
All the possible errors are listed in Table 11 (page 40).

Example
None

mCO_DictSetRet
This function is used to set the return status of a dictionary operation. This is only used within an object handling function.

Syntax
void mCO_DictSetRet(unsigned char retVal)

Parameters
unsigned char retVal: The return status of the object request. All the possible errors are listed in Table 11 (page 40).

Return Values
None

Example
Refer to the code in Example 9 (page 42).
ECAN™ DRIVER

The functions in this section describe the functional interface of the ECAN driver. Note that the driver provided with the CANopen Stack has been specifically designed for PIC18F devices with ECAN technology. It is also possible to use an external CAN controller, and therefore a different driver with different function calls. In this event, the user will need to provide an appropriate driver.

mCANEventManager

This is an event handling function. All queued events are processed from within this function. This function is called within the CANopen Stack when CO_ProcessAllEvents is called.

Syntax

```c
void mCANEventManager(void)
```

Parameters

None

Return Values

None

Example

None

mCANReset

This function resets CAN communications and sets the appropriate bit rate. This function is called from within the CANopen Stack when a Reset request is received either from the application or the NMT master.

Syntax

```c
void mCANReset(unsigned char CANBitRate)
```

Parameters

None

Return Values

None

Example

None.

mCANOpenComm

This function opens CAN communications. This function should be treated as a request. Depending on the bus activity, communications may not be opened immediately.

Syntax

```c
void mCANOpenComm(void)
```

Parameters

None

Return Values

None

Example

None
mCANCloseComm
This function closes CAN communications.

Syntax
void mCANCloseComm(void)

Parameters
None

Return Values
None

Example
None

mCANIsCommOpen
This function can be used to query the driver to determine if communications are opened or closed.

Syntax
BOOL mCANIsCommOpen(void)

Parameters
None

Return Values
TRUE: Communications are opened.
FALSE: Communications are closed.

Example
None.

mCANErrIsOverFlow
This function is used to query the driver for a receive buffer overflow condition. If an overflow condition is found then the condition can be removed by calling the mCANErrClearOverFlow function. When an overflow condition has happened one or more messages have been lost. How this is handled depends on the application; the specification does not require a particular method for handling this condition.

Syntax
void mCANErrIsOverFlow(void)

Parameters
None

Return Values
TRUE: A receive buffer has overflowed.
FALSE: A receive buffer has not overflowed.

Example
None
mCANErrClearOverFlow
Remove the receive buffer overflow condition.

Syntax
void mCANErrClearOverFlow(void)

Parameters
None

Return Values
None

Example
None

mCANSetBitRate
This function sets the current bit rate. The bit rate is not changed immediately; it is actually queued in the driver until the
driver and CAN hardware are ready to accept a change. Typically this is only called once at start-up.

Syntax
void mCANSetBitRate(unsigned char CANBitRate)

Parameters
unsigned char CANBitRate: This can be any value; however, only values 0 through 8 are considered valid. All other
values will automatically default to the bit rate identified by option 0. All 9 options are defined in the file CO_DEFS.DEF.

Return Values
None

Example
None

mCANGetBitRate
This function returns the current bit rate used by the driver.

Syntax
unsigned char mCANGetBitRate(void)

Parameters
None

Return Values
unsigned char: The current bit rate. Only values 0 through 8 are valid; however, the function may return other values if
mCANSetBitRate() was passed a value other than the valid values.

Example
None
**mCANOpenMessage**

This function scans the available mailbox space for an open slot. The CAN identifier must be passed in along with a unique non-zero handle to that identifier. If a slot is found then all messages containing the provided CAN identifier will be received and the handle will be used to identify the message. The handle will also be returned to the caller if found; otherwise, the return will be zero. The calling function must maintain the handle if the endpoint is to be released at a later time without a Reset.

The CAN identifier is added but not activated until the bus and the driver are ready. In future CAN modules this queuing functionality may be removed, depending on available hardware support.

**Syntax**

```c
void mCANOpenMessage(unsigned char MsgTyp, unsigned long COBID, unsigned char hRet)
```

**Parameters**

- `unsigned char MsgTyp`: The unique handle to the identifier. It must be non-zero.
- `unsigned long COBID`: The CAN identifier of the message to be allowed.

**Return Values**

- `unsigned char hRet`: The return status. This will be either 0 or the handle.

**Example**

None

**mCANCloseMessage**

This function scans the mailbox space for the handle. If found, the CAN identifier is removed from the receive list.

The CAN identifier is only queued to be removed from the list. Thus messages may still be received until the driver can fully remove the CAN identifier from the hardware. In future CAN modules this queuing functionality may be removed depending on hardware support.

**Syntax**

```c
void mCANCloseMessage(unsigned char hMsg)
```

**Parameters**

- `unsigned char hMsg`: The handle to the message.

**Return Values**

None

**Example**

None
mCANIsGetRTR
This function queries the driver for the RTR condition of the current message. The function mCANIsGetReady should be called prior to this request to set the current message.

Syntax
void mCANIsGetRTR(void)

Parameters
None

Return Values
None

Example
None

mCANIsGetReady
This function scans for a receive event. If found, it places a handle associated to the receive buffer into an internal register which can be accessed by mCANFetchRetStat. Otherwise, it returns zero. If a valid message is waiting, it should be processed prior to calling the function again.

Buffer access on successive receive related calls is assumed, i.e., the handle is not required for associated read functions. For example, calls to mCANGetDataLen() and mCANGetDataByteN() functions assume the most current received message data is being requested.

Syntax
void mCANIsGetReady(void)

Parameters
None

Return Values
None

Example
None

mCANReadMessage
Calling this function indicates to the driver that the current message has been processed, and the driver is now free to use the buffer for a new message. The function mCANIsGetReady should have been called prior to this request to set the current message.

Syntax
void mCANReadMessage(void)

Parameters
None

Return Values
None

Example
None
mCANGetPtrRxCOB
This function retrieves the pointer to the current identifier. It also points to the whole message stored in Microchip format.

Syntax
unsigned char * mCANGetPtrRxCOB(void)

Parameters
None

Return Values
unsigned char *: Returns a pointer to the received CAN identifier.

Example
None

mCANGetPtrRxData
This function retrieves the pointer to the current data.

Syntax
unsigned char * mCANGetPtrRxData(void)

Parameters
None

Return Values
unsigned char *: Returns a pointer to the received data.

Example
None

mCANGetDataLen
This function retrieves the length of the current message or RTR request.

Syntax
unsigned char mCANGetDataLen(void)

Parameters
None

Return Values
unsigned char: Length of message or RTR request.

Example
None
mCANGetDataByten
This represents a total of eight functions, where the trailing \( n \) can represent values from 0 to 7. Each will return the corresponding data byte of the message received.

Syntax
unsigned char mCANGetDataByten(void)

Parameters
None

Return Values
unsigned char: The data byte.

Example
None

mCANIsPutReady
This function scans for an available output buffer. If successful, the handle passed is the same as the handle returned; otherwise a zero is returned. The function mCANFetchRetStat must be called to get the return value.

Syntax
void mCANIsPutReady(putHndl)

Parameters
unsigned char putHndl: The handle of the message.

Return Values
None

Example
None

mCANIsPutFin
This function queries the driver for any message that has been placed on the bus and returns the handle to the message that was sent. The function mCANFetchRetStat must be used to get the handle to the message.

This function should only be called one time for a transmit indication. Calling this function a second time after receiving an indication may not return the same handle.

Syntax
void mCANIsPutFin(void)

Parameters
None

Return Values
None

Example
None
mCANSendMessage
This function is used to indicate to the driver that the data, length, and CAN identifier have been loaded and are ready to be sent.

Syntax
void mCANSendMessage(void)

Parameters
None

Return Values
None

Example
None

mCANGetPtrTxCOB
This function gets the pointer to the transmit CAN identifier buffer.

Syntax
unsigned char * mCANGetPtrTxCOB(void)

Parameters
None

Return Values
unsigned char *: The pointer to the CAN identifier transmit buffer.

Example
None

mCANGetPtrTxData
This function gets the pointer to the transmit data buffer.

Syntax
unsigned char * mCANGetPtrTxData(void)

Parameters
None

Return Values
unsigned char *: A pointer to the data transmit buffer.

Example
None
mCANPutDataLen
This function sets the data length or the RTR request length.

Syntax
void mCANPutDataLen(unsigned char CANlen)

Parameters
unsigned char CANlen: The length or the RTR request length of the message.

Return Values
None

Example
None

mCANPutDataByten
This represents a total of eight functions, where the trailing $n$ represents values from 0 to 7. Each can be used to set the corresponding byte to be sent.

Syntax
void mCANPutDataByten(unsigned char CANDat)

Parameters
unsigned char CANDat: Data byte.

Return Values
None

Example
None

mCANFetchRetStat
This function is used to get the status of a function that returns status. The functions that return status are noted.

Syntax
unsigned char mCANFetchRetStat(void)

Parameters
None

Return Values
unsigned char: The status of the last operation.

Example
None
FINISHING THE APPLICATION

Of course there are still some CAN specific details that need to be handled. Here are some points to remember:

- Objects: Define and develop all objects and handling functions and link them to the dictionary. Objects that are defined by a function require of course extra coding because of the handling function; however, these types of objects are highly flexible.
- Dictionary: Place all objects within their proper place within the dictionary. Properly define the control, length, and the reference information for the objects.
- PDOs: These still must be defined and developed. Remember that PDOs can be static or dynamic; static methods will always be code and process-efficient but are obviously not flexible like dynamic PDOs. There are also a number of PDO transmission types that depend on the specific application. For these reasons, only a base set of tools are provided so the designer can develop the most efficient code for the application.
- Timing: Provide a time base by using one of the timers or some external time source.
- Initialization: Develop proper initialization code. Many objects need to be initialized from some static source such as ROM, EEPROM, or even switches connected to input pins.
- Main Processing: Develop efficient cooperative design practices in order to properly capture and handle all events.
- Events: There are numerous events. Ensure proper handling is in place where necessary. For example, Reset requests from the network are provided as events to the application. It is left up to the application designer to decide how to handle a Reset request.
- Compile Time Setup: Set up the appropriate compile time options to achieve optimal resource usage and efficiency.

RESOURCE USAGE

Device resources used by the stack are highly dependent on the compile time options, as well as compiler optimizations. The application designer should expect the stack to consume about 7000 to 10,000 bytes of program memory and 300 bytes of data memory with optimization.

Using all of the optimizations available in the MPLAB® C18 Compiler (v2.30.01), the demonstration application provided with this application note requires 7434 bytes of program memory and 314 bytes of data memory.

CONCLUSION

Developing a CANopen device can be an arduous task. By using the CANopen Stack and its tools, a good portion of the work is already accomplished by removing much of the CANopen and CAN specific communications management. This allows the applications designer to focus a much greater percentage of his or her effort on the application, and less on the specifics of CANopen.

REFERENCES


APPENDIX A: SOFTWARE DISCUSSED IN THIS APPLICATION NOTE

Because of the number of individual modules and their size, a complete source code listing of the CANopen Stack is not provided here. Interested users are invited to download the .zip archive file, including all source and header files, from the Microchip corporate web site at:

www.microchip.com
Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip’s Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip’s code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.
WORLDWIDE SALES AND SERVICE

AMERICAS
Corporate Office
2355 West Chandler Blvd.
Chandler, AZ  85224-6199
Tel:  480-792-7200
Fax:  480-792-7277
Technical Support: 480-792-7627
Web Address: www.microchip.com

Atlanta
3780 Mansell Road, Suite 130
Alpharetta, GA  30022
Tel:  770-640-0034
Fax:  770-640-0070

Boston
2 Lan Drive, Suite 120
Westford, MA  01886
Tel:  978-692-3848
Fax:  978-692-3821

Chicago
333 Pierce Road, Suite 180
Itasca, IL  60143
Tel:  630-285-0071
Fax:  630-285-0075

Dallas
16200 Addison Road, Suite 255
Addison Plaza
Addison, TX  75001
Tel:  972-818-7423
Fax:  972-818-2924

Detroit
Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI  48334
Tel:  248-538-2250
Fax:  248-538-2260

Kokomo
2767 S. Albright Road
Kokomo, IN  46902
Tel:  765-864-8360
Fax:  765-864-8387

Los Angeles
25950 Acero St., Suite 200
Mission Viejo, CA  92691
Tel:  949-462-9523
Fax:  949-462-9608

San Jose
1300 Terra Bella Avenue
Mountain View, CA  94043
Tel:  650-215-1444
Fax:  650-961-0286

Tokyo
6285 Northam Road, Suite 108
Mississauga, Ontario L4V 1X5, Canada
Tel:  905-673-0699
Fax:  905-673-6509

ASIA/PACIFIC
Australia
Microchip Technology Australia Pty Ltd
Unit 32 41 Rawson Street
Epping 2121, NSW
Tel:  61-2-9868-6733
Fax:  61-2-9868-6755

China - Beijing
Unit 706B
Wan Tai Bei Hai hdq.
No. 6 Chaoyangmen Bei Str.
Beijing, 100027, China
Tel:  86-10-85282100
Fax:  86-10-85282104

China - Chengdu
Rm. 2401-2402, 24th Floor,
Ming Xing Financial Tower
No. 88 TIDU Street
Chengdu 610016, China
Tel:  86-28-88768200
Fax:  86-28-88766599

China - Fuzhou
Unit 28F, World Trade Plaza
No. 71 Wusi Road
Fuzhou 350001, China
Tel:  86-591-75035056
Fax:  86-591-75035211

China - Hong Kong SAR
Unit 901-6, Tower 2, Metropiazza
233 Hing Fong Road
Kwai Fong, K.N.T., Hong Kong
Tel:  852-2401-1200
Fax:  852-2401-3431

China - Shanghai
Room 701, Bldg. B
Fer East International Plaza
No. 317 Xian Xia Road
Shanghai, 200051
Tel:  86-21-6275-5700
Fax:  86-21-6275-5060

China - Shenzhen
Rm. 1812, 18/F, Building A, United Plaza
No. 5022 Binhe Road, Futian District
Shenzhen 518033, China
Tel:  86-755-82901380
Fax:  86-755-8295-1393

China - Qingdao
Rm. B505A, Fullhope Plaza,
No. 12 Hong Kong Central Rd.
Qingdao 266071, China
Tel:  86-532-50273535  Fax:  86-532-5027205

India
Divyasee Chambers
1 Floor, Wing A (A3/A4)
No. 11, O'Shaugnessy Road
Bangalore, 560 025, India
Tel:  91-80-22290061  Fax:  91-80-22290062

Japan
Yusen Shin Yokohama Building 10F
3-17-2, Shin Yokohama, Kohoku-ku,
Yokohama, Kanagawa, 222-0033, Japan
Tel:  81-45-471-6166  Fax:  81-45-471-6122

Korea
1681-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea 135-882
Tel:  82-2-554-7200  Fax:  82-2-558-5932 or 82-2-558-5934

Singapore
200 Middle Road
#07-02 Prime Centre
Singapore, 188980
Tel:  65-6334-8870  Fax:  65-6334-8850

Taiwan
Kaochisun Branch
30F - 1 No. 8
Min Chuan 2nd Road
Kaochisun 806, Taiwan
Tel:  886-7-536-4816
Fax:  886-7-536-4817

Taiwan
Taiwan Branch
11F-3, No. 207
Tung Hua North Road
Taipei, 105, Taiwan
Tel:  886-2-2717-7175  Fax:  886-2-2545-0139

Taiwan
Taiwan Branch
13F-3, No. 295, Sec. 2, Kung Fu Road
Hsinchu City 300, Taiwan
Tel:  886-3-572-9526
Fax:  886-3-572-6459

EUROPE
Austria
Durisistrasse 2
A-4600 Wels
Tel:  31-4420-9895  Fax:  31-4420-9910

Denmark
Regus Business Centre
Lastrup høj 1-3
Ballerup DK-2750 Denmark
Tel:  33-1-69-53-63-20
Fax:  33-1-69-30-90-79

France
Parc d’activite du Moulin de Massy
43 Rue du Saule Trapu
Batiment A - 1er Etage
91300 Massy, France
Tel:  33-1-69-53-63-20
Fax:  33-1-69-30-90-79

Germany
Sternheilstrasse 10
D-85737 Ismaning, Germany
Tel:  44-99-627-144-0  Fax:  49-99-627-144-44

Italy
Via Salvatore Quasimodo, 12
20025 Legnano (MI)
Tel:  39-0331-742611  Fax:  39-0331-466781

Netherlands
Waagenburghplein 4
NL-5152 JR, Drunen, Netherlands
Tel:  31-416-690399

United Kingdom
505 Eskdale Road
Winneresh Triangle
Wokingham Berkshire, England RG41 5TU
Tel:  44-118-921-5869  Fax:  44-118-921-5820

07/12/04