Motion Engineering
33 South La Patera Lane
Santa Barbara, CA 93117-3214
ph (805) 681-3300
fax (805) 681-3311
tech@motioneng.com
www.motioneng.com

APPLICATION NOTE 9D00-0162, REV B

(ECO 1621)

The XMP CANOpen Interface

Introduction

CANOpen is an international standard that defines an industrial network of connecting a wide variety of nodes together.

A CANOpen interface is now available on several of MEI's XMP products. This application note describes how the CANOpen interface is implemented in the MPI and provides the user information on configuring and running a CAN I/O network.

Document Revision History				
Rev.	**Date**	**Description**	**DCR No.**	**DCO No.**
B	20May2002	Updated to match 20020403.1.3 MPI version.	616	1621
A	21Feb2002	Created.	598	1404

Hardware

CANOpen is a serial network that uses a bus topology. The CANOpen bus always contains two signal wires, CAN+ and CAN-, which carry the differential serial data and a ground (GND). It is also common for most CANOpen nodes to provide a shield connection.

Similar to most industrial buses, the signal wires need to be terminated. CANOpen requires a 120Ω resistor at both ends of the main bus. If these resistors are not fitted then the network will not function properly. Some node suppliers build the terminating resistor into the node and provide a jumper or switch to enable it. You will need to check your nodes' data sheets for the inclusion of a terminating resistor.

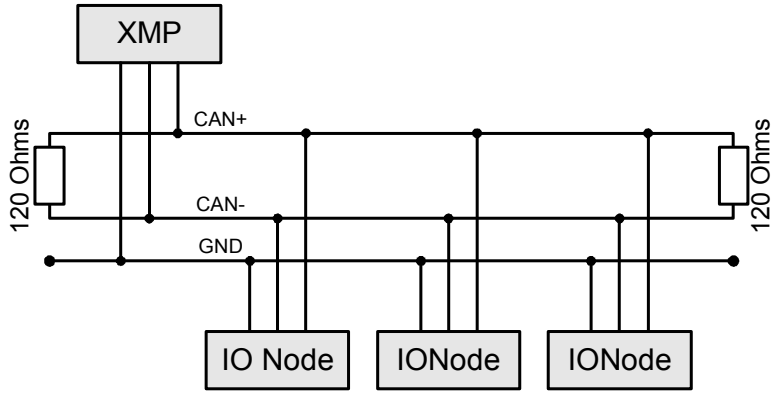
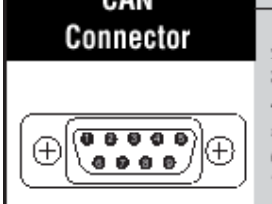


Figure 1: CANOpen Network

The pin out for the XMP's CAN D9 connector is:



Pin	Signal	Description
1	–	Reserved
2	CAN_L	CAN_L bus line dominant low
3	CAN_GND	CAN Ground
4	–	Reserved
5	(CAN_SHLD)	Optional CAN Shield
6	GND	Optional Ground
7	CAN_H	CAN_H bus line dominant high
8	–	Reserved
9	(CAN_V+)	Optional CAN external supply

Table 1: CAN Connector Pin Out

A CANOpen node either has an opto-isolated or un-isolated interface. The use of opto-isolation is primarily provided as an EMC countermeasure and is used to cope with potential differences in the ground. These effects are more pronounced for large machines and cable lengths. Therefore, the use of opto-couplers is recommended for bus lengths greater that 200m. The disadvantage of opto-couplers is that they reduce the maximum permissible bus length for a given bit rate.

The XMP CAN interface is available with or without opto-isolation. This option needs to be specified at the time your XMP is ordered.

Most types of nodes require a separate power supply to drive the local logic and the I/O interfaces. For nodes that use opto-isolated network interfaces between +7 to +24V, it is also required to power the interface circuitry.

Each node on the network must have a unique node number, in the range of 1 to 127. The node number is commonly set with a bank of DIP switches on each node. If two nodes are given the same node number, network errors are generated and unpredictable problems will be encountered. The node number of the XMP can be changed from the factory default of 1 using the **meiCanConfigSet** function. See meiCanConfigSet for more info.

In order for all nodes to communicate they must all use the same bit rate. Normally the bit rate that a node uses is set by DIP switches. If all of the nodes on a CANOpen network do not use the same bit rate then the whole network or some of the nodes on the network will not work properly. The bit rate of the XMP is set via software (**meiCanConfigSet**). See meiCanConfigSet for more info.

XMP Overview

In the example below, the XMP uses a dedicated CAN processor to handle the network. This ensures that the motion will not be affected by the CAN network.

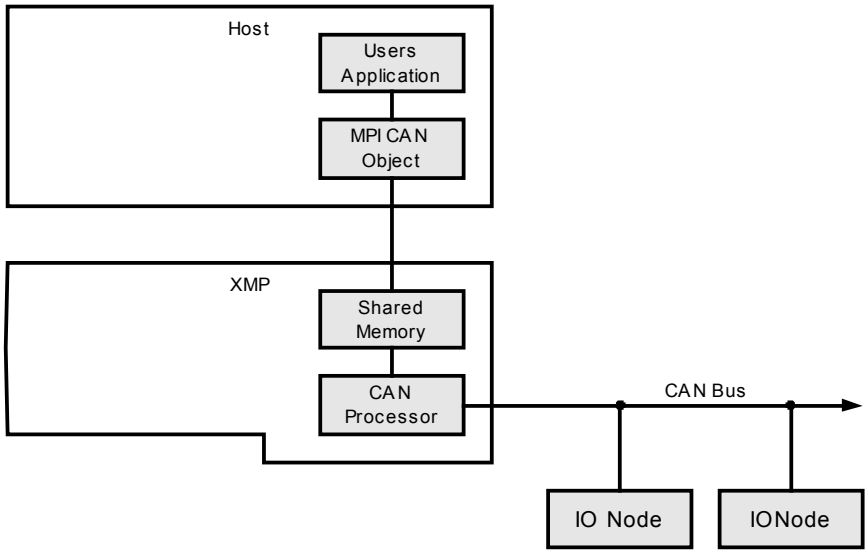


Figure 2: Overview of XMP architecture and CAN network

The XMP operates as a master node on the network with all the IO nodes being slaves. This arrangement implies that there may only be one XMP on any CAN Network.

Supported Nodes

The XMP CANOpen interface is designed to support any CANOpen node conforming to CANOpen I/O Node Profile, DS401 version 2.0. The interface has been fully tested with nodes from the following manufacturers: Beckhoff, Wago, and Selectron.

Software

Utilities

Motion Console

The Motion Console interface has changed slightly in order to accommodate support for CAN I/O. Three new buttons/windows have been added to Motion Console.

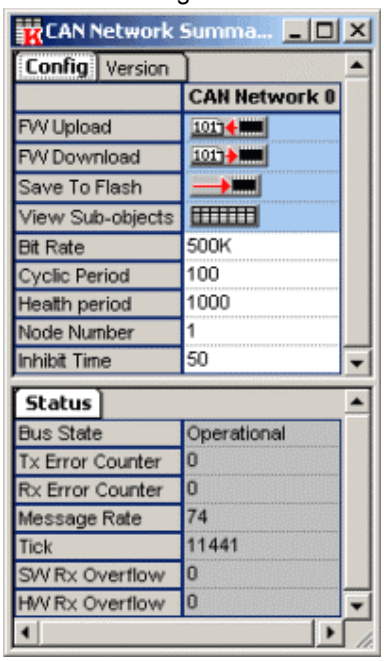
 - This button will open the **CAN Network Summary** window.

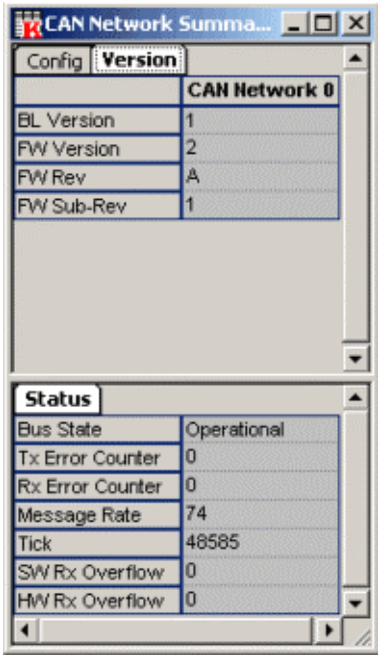
 - This button will open the **CAN Node Summary** window.

 - This button will open the **I/O: CAN** window.

 **CAN Network Summary**

This Config window displays the user configurable parameters of CAN, whereas the Version window is not user-configurable.





Config

- **FW Upload** – allows the user to get a copy of the current CAN controller's firmware.
- **FW Download** – allows the user to upgrade the CAN controller's firmware.
- **Save to Flash** – saves the current flash configuration the CAN sun-system is using.
- **View Sub-objects** –

- **Bit Rate** – see *Table 2: CANOpen Bit Rates*.
- **Cyclic Period** – the period between sending consecutive SYNC messages (ms). A value of zero will disable the SYNC messages from being produced.
- **Health Period** – the period used for checking the health of nodes (ms).
- **Node Number** – the node number of the XMP on the CAN network.
- **Inhibit Time** – this coefficient defines the minimum time between two successive PDO messages.

Status

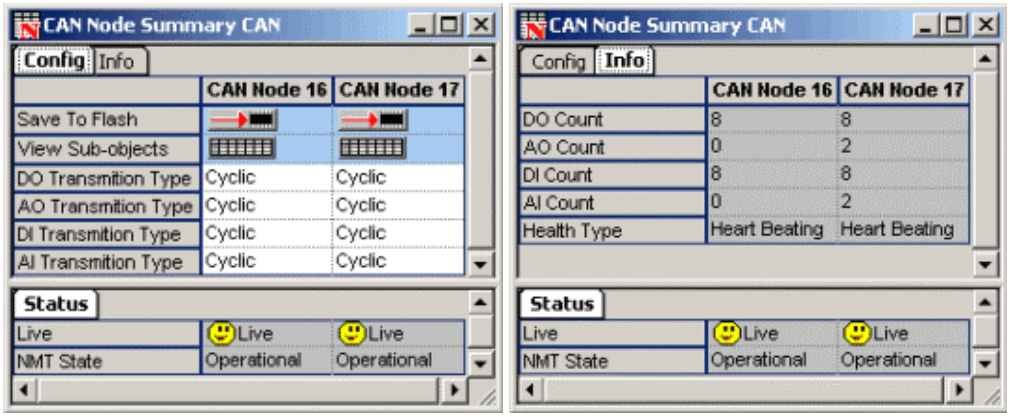
- **Bus State** – the CAN bus will be in one of the following states: off, operational, or passive.
- **Tx Error Counter** – the current value of the transmit error counter.
- **Rx Error Counter** – the current value of the receive error counter.
- **Message Rate** – allows the user to specify the Node Guard and Heartbeat times for the health protocols.
- **Tick** – A Counter that is incremented every 1ms.
- **SW Rx Overflow** –
- **HW Rx Overflow** –

Version

- **BL Version** – the version number of the CAN bootloader.
- **FW Version** – the CAN firmware version number.
- **FW Rev** – the CAN firmware revision number.
- **FW Sub-Rev** – the CAN firmware sub-revision number.

Can Node Summary

This Config window displays the user configurable parameters of CAN, whereas the Info window is not user-configurable.



Config

- **Save to Flash** – saves the current flash configuration the CAN sun-system is using.
- **View Sub-objects** –
- **DO Transmission Type** – the current state of the digital output bit on the specified CAN node.
- **AO Transmission Type** – the current state of the analog output bit on the specified CAN node.
- **DI Transmission Type** – the current state of the digital input bit on the specified CAN node.
- **AI Transmission Type** – the current state of the analog input bit on the specified CAN node.

Info

- **DO Count** – the number of digital outputs supported on this node. The CANOpen protocol only allows the number of digital inputs and outputs to be interrogated in multiples of eight, i.e. if a node has one digital output the “digitalOutputCount” will return eight.
- **AO Count** – the number of analog outputs supported on this node.
- **DI Count** – the number of digital inputs supported on this node. The CANOpen protocol only allows the number of digital inputs and outputs to be interrogated in multiples of eight, i.e. if a node has one digital output the “digitalOutputCount” will return eight.
- **AI Count** – the number of analog inputs supported on this node.
- **Health Type** – The CANOpen protocol being used to check the health of this node.

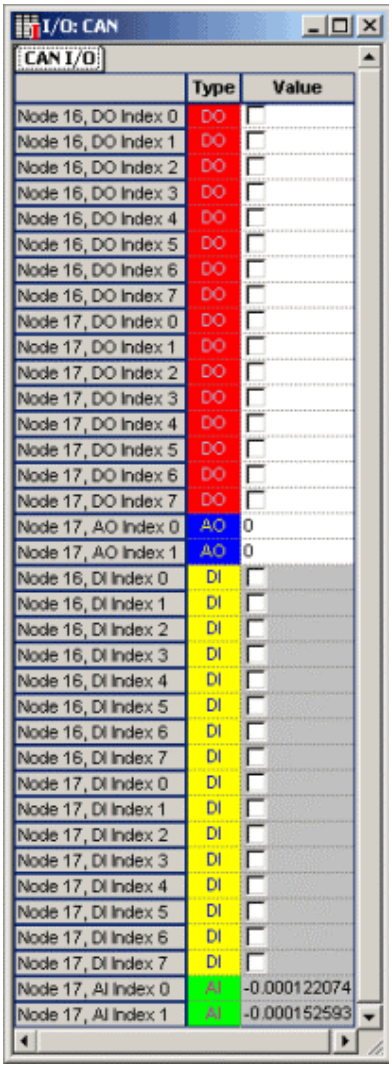
Status

- **Live** – the system will either be “live” or “dead.”
- **NMT State** – CANOpen protocol Network Management state

If there is an error in the system (i.e. loss of power, disconnected cable), the face icons in the Status window will change to blue to symbolize a “dead” controller.

I/O CAN

This window displays a breakdown of the I/O for a CAN system.

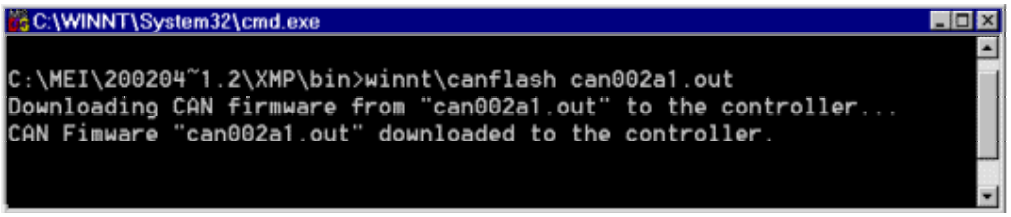


CAN I/O

- DO, AO, DI, AI – The node types are color coded by type: Digital Output, Analog Output, Digital Input and Analog Input.

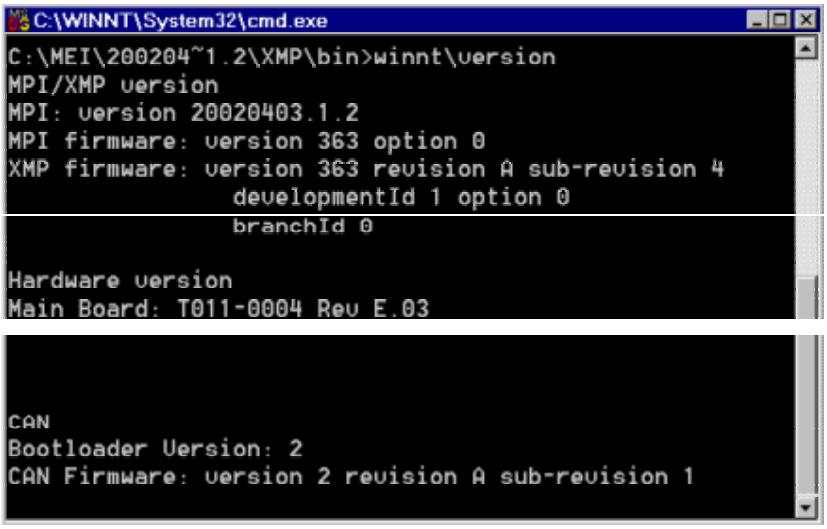
CANflash.exe

The CANflash utility can be used to download CAN firmware to the CAN controller. Use the "-?" flag for additional usage instructions. Please see the screenshot below for an example.



Version.exe

The standard MPI Version utility has been expanded to include CAN Bootloader Version and CAN Firmware Version. Please see the screenshot below for an example.



MPI CAN Object

The MPI has been extended to allow the user easy access to the I/O nodes connected to the CANOpen interface of an XMP. A new MPI object has been introduced to encapsulate this functionality. Please see the **CAN Object Methods** and **CAN Object Data Types** sections in this document for details of all the functions and data types that can be used with the CAN object.

If an XMP controller does not support the CANOpen interface, the meiCanValidate function will return MEICanMessageINTERFACE_NOT_FOUND.

The CAN system uses the MEICanConfig and MEICanNodeConfig structures to hold all of the user configurable quantities. These structures are stored in non-volatile flash memory. When the XMP is released from reset (normally soon after the host powers up or after a call to mpiControlReset), the CAN Processor will initialize itself with data from MEICanConfig and MEICanNodeConfig before starting to scanning the network for nodes.

The functions meiCanConfigGet, meiCanConfigSet, meiCanNodeConfigGet and meiCanNodeConfigSet allow the user to modify the current configuration of the CAN Processor, and meiCanFlashConfigGet, meiCanFlashConfigSet,

meiCanFlashNodeConfigGet and meiCanFlashNodeConfigSet functions allow the user to modify the configuration that the CAN system will use after the next reset.

The MEICanVersion structure returns the version information about the CAN system on the XMP.

After the CAN processor has finished scanning the network, it will have completed the MEICanNodeInfo structures for each node. The user can call the meiCanNodeInfo function to query this initial configuration for each of the nodes.

Configuring the CAN System

Bit Rate

The CANOpen standard defines a set of bit rates (see Table 2: CANOpen Bit Rates) that can be supported. Any CANOpen node must support at least one of these bit rates. All the nodes on the CAN network must be operating at the same bit rate. Any of these standard bit rates can be used with the XMP.

Due to the electrical characteristics of a CAN network, the maximum length of a CAN network (and the corresponding drop lengths) is dependent upon the bit rate that is chosen. See Table 2: CANOpen Bit Rates below.

Bit Rate	Maximum Bus Length (m)	Maximum Drop Length (m)	Maximum Cumulative Drop Length (m)
1M	25*	2	10
800k	50*	3	15
500k	100	6	30
250k	250	12	60
125k	500	24	120
50k	1000	60	300
20k	2500	150	750
10k	5000	300	1500

*No opto-isolation

Table 2: CANOpen Bit Rates

Transmission Types

The XMP CANOpen interface uses four messages (serial packets of data on the CAN bus) to pass IO data between the XMP and an IO node. Each message contains either the digital input, digital output, analog input, or analog output data.

The XMP supports two standard communication methods to transmit IO data between the XMP and each of the IO nodes—cyclic transmission and event transmission. For most applications cyclic messaging (the default) will be sufficient but the transmission type fields within the MEICanNodeConfig structure allow the user to select an alternative transmission type for each of the IO messages going to and from a node.

Cyclic Transmission

The Cyclic Transmission type, shown in Figure 3: Cyclic Transmission, transfers IO data messages between the XMP and the nodes using a cyclic protocol. The trigger for each cycle is a synchronization message that is transmitted at a regular rate by the XMP. When a

node receives the synchronization message, it latches and transmits the current state of its inputs. Also immediately after receiving the synchronization message, the master also transmits command messages to all the nodes with their new output states, which will get applied on the next synchronization message. An idle period is also needed to allow time for any non-cyclic messages to be transmitted.

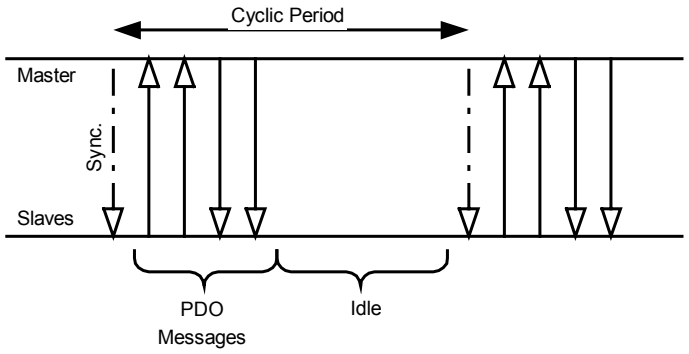


Figure 3: Cyclic Transmission

The advantage of this scheme is that it generates a predicable loading of data on the bus. The latency on transmitted data is predictable, but the latency it is not the absolute minimum that can be achieved.

Cyclic Period

The *cyclicPeriod* field within the **MEICanConfig** structure allows the user to specify the period in milliseconds that the XMP will use between the successive transmission of synchronization messages. The minimum cyclic period that can be used is dependent upon the chosen bit rate and the number of nodes. Assuming that all the nodes have inputs and outputs that are analog and digital, the minimum cyclic period that can be used is given in the following table.

Bite Rate	<5 Nodes	<10 Nodes	<50 Nodes	<128 Nodes
1M	3	5	30	60
800k	3	6	30	80
500k	5	10	50	200
250k	10	18	89	300
125k	19	36	200	500
50k	46	90	500	2000
20k	200	300	2000	3000
10k	300	500	3000	6000

Table 3: Minimum Cyclic Period (ms) For Fully Featured IO Nodes

Event Transmission

The Event Transmission type, shown in Figure 4: Event Transmission, only transmits IO data messages when an “event” occurs on the source node (either the XMP or the IO node) to change the IO data. The events that force the transmission are a new state of an input is detected on an IO node or a new output state is commanded on the XMP.

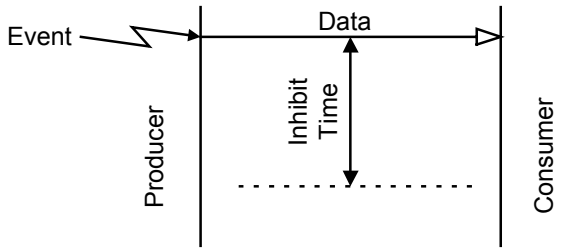


Figure 4: Event Transmission

The advantage of this type of messaging is that short reaction times are attainable, but this is accomplished at the expense of variable network traffic, and the possibility of saturating the network. In many cases, the reaction time is not significant in relation to other time delays in the system, e.g. the user's application or delays in task switching.

Inhibit Time

If the source node's events occur at a very fast rate, the number of messages generated can swamp the network blocking out other messages. To prevent an excess of messages, nodes can optionally support inhibit times for their transmit PDOs. This value defines the minimum time between two successive PDO messages.

The inhibitTime field within the **MEICanConfig** structure allows the user to specify the period in milliseconds that all nodes on the network will use. A reasonable inhibit time is half a cyclic period.

CAN Status

CANOpen and the underlying CAN protocols provide a set of error detection schemes that display useful diagnostic and warning data.

CAN Bus State

All CAN hardware maintains two error counters that are increased when transmit or receive errors are detected, and decreased when successful transmissions or receptions are achieved. In an error free operational system, these counters should be zero. The magnitude of these counters control the following state machine:

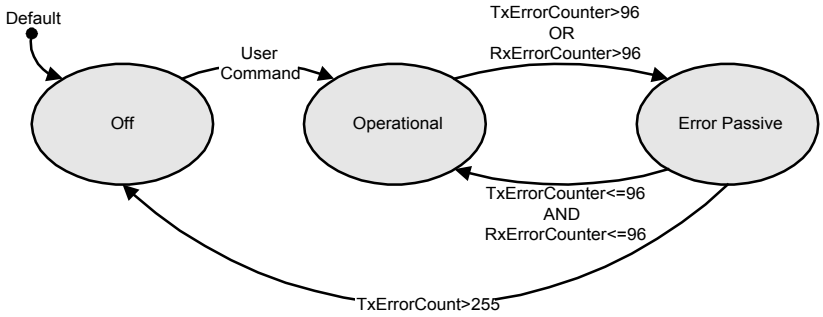


Figure 5: CAN Controller State

When a node is in its **Operational** state it will participate fully with all communications over the network, as the errors increase the CAN hardware will become **Passive** (detecting errors but not generating error messages), before turning **Off and** isolating the node from the network. This feature allows nodes that are either malfunctioning or mis-configured to be isolated for the network, thereby allowing the remaining nodes to successfully communicate.

The magnitude of the two counters and the bus state for the XMP's interface to the CAN network can be monitored using corresponding fields within the **MEICanStatus** structure.

Node Health

All networks including CAN are vulnerable to faults such as breaks in the bus wiring or loss of power by some of the nodes. CANopen defines two methods for the master node (the XMP in our case) to periodically check the presence of nodes on the network—node guarding, and heart beating.

Using these services the XMP can monitor the health of the communications to each of the nodes. The current health of each node is reported in the *live* field of the **MEICANNodeStatus** structure.

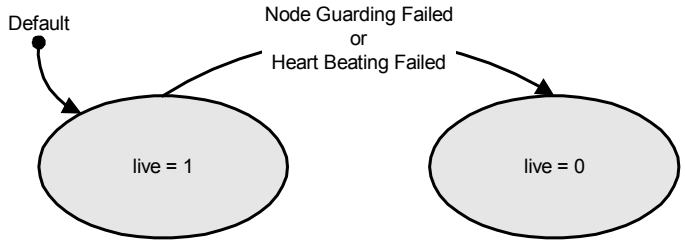


Figure 6: Node Health States

It is mandatory for a node to support either or both the guarding and heart beating protocols. The heartbeat protocol has only recently been introduced to CANOpen (in June 1999), and will probably not be supported on many nodes but its adoption is recommended for all new nodes. The XMP's implementation will operate with either protocol, automatically detecting the protocol that each node supports and using the most appropriate. The *healthType* field of the **MEICanNodeInfo** structure reports the health checking protocol being used with each node.

Node Guarding protocol

The Node Guarding protocol has the master sending an RTR message to all nodes on the network and checks to see whether a response is received from each of the nodes.

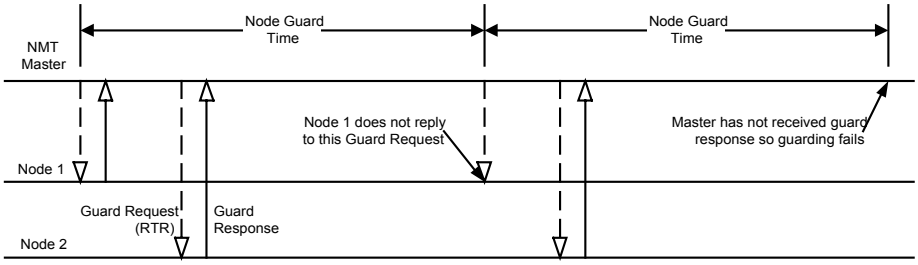


Figure 7 Node Guarding

Heart Beating protocol

In the Heart Beating protocol, each node periodically broadcasts a heartbeat message. The period between transmitting the heartbeat messages is half the health period. If the XMP does not receive a message within a specific time window, it generates a heartbeat error for that node.

The advantage of the Heart Beating protocol over the Node Guarding protocol is that the number of messages is reduced in half, thereby freeing up bandwidth for other messages.

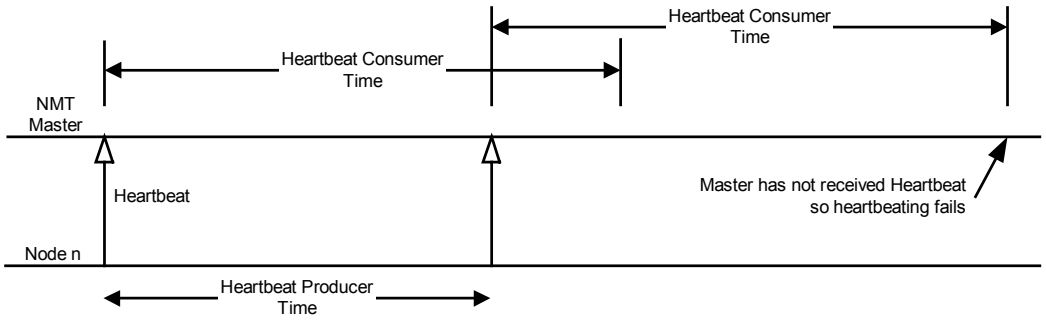


Figure 8 Heart Beating

Health Period

The **healthPeriod** field of the **MEICanConfig** structure allows the user to specify the Node Guard and Heartbeat times for the health protocols according to the following table. The same period is used for all nodes.

Protocol times	Value
Node Guard Time	healthPeriod
Heartbeat Producer Time	healthPeriod / 2
Heartbeat Consumer Time	healthPeriod

Table 4: Node Heath times

For most applications it is recommended that the healthPeriod should be set to ten times the cyclic period.

Emergency Messages

Every type of CANOpen node can transmit an emergency message. These messages are designed to report errors and warnings, as well as fatal problems on a node. The contents of these emergency messages are very dependent upon the manufacturer and node type. To interpret this data, you will need to refer to the node manufacture's data. If an emergency message is generated by a node the event handling scheme described in the events section below allows the user's application to receive the emergency message data.

Using the I/O

The CAN object provides four sets of functions to access the IO that may be present on a node. The XMP's CAN system supports digital inputs and outputs, in addition to analog inputs and outputs. CANOpen nodes may contain combinations of all of these IO types up to a maximum of 64 digital inputs, 64 digital outputs, 8 analog inputs, and 8 analog outputs per node.

The following functions operate on individual inputs or outputs:

meiCanNodeDigitalInputGet

meiCanNodeDigitalOutputGet
meiCanNodeDigitalOutputSet

meiCanNodeAnalogInputGet

meiCanNodeAnalogOutputGet
meiCanNodeAnalogOutputSet

The following functions operate on all the digital inputs or outputs on a specified node:

meiCanNodeDigitalInputsGet

meiCanNodeDigitalOutputsGet
meiCanNodeDigitalOutputsSet

All analog data that is handled by the interface are double numbers scaled to between ±1.0.

Handling Events

The CAN interface on the XMP generates many different types of asynchronous events. These events are:

- A change in the XMP's bus state.
- A change in a node's health.
- A change in the state of an input node's analog or digital inputs.
- A node transmitted an emergency message.
- A node has transmitted a boot message.
- The XMP CAN firmware detected a lost message.

To allow a user's program to respond to these events, they have been appended to the standard MPI event handling scheme. Figure 9 shows an overview of how events are relayed to the users application.

1. The CANOpen firmware detects one of the CAN events.
2. There is a mask within the XMP firmware that allows only a specified set of events to reach the host. This mask is interrogated and modified with the **meiCanEventNotifyGet** and **meiCanEventNotifySet** functions.
3. Like all other events in the MPI, the user must install an Event Manager on the host. You will find the **serviceCreate** and **serviceDelete** functions from apputils convenient for installing an Event Manager.
4. For each thread that needs to know about CAN events, the user will need to create a notify object, specifying a mask for the required events.
5. The user's application can use the **mpiNotifyEventWait** function to either poll or wait for a CAN event to be generated. A valid event returned from **mpiNotifyEventWait** may also contain extra field of information relevant to the event produced. e.g. the new bus state or node number.

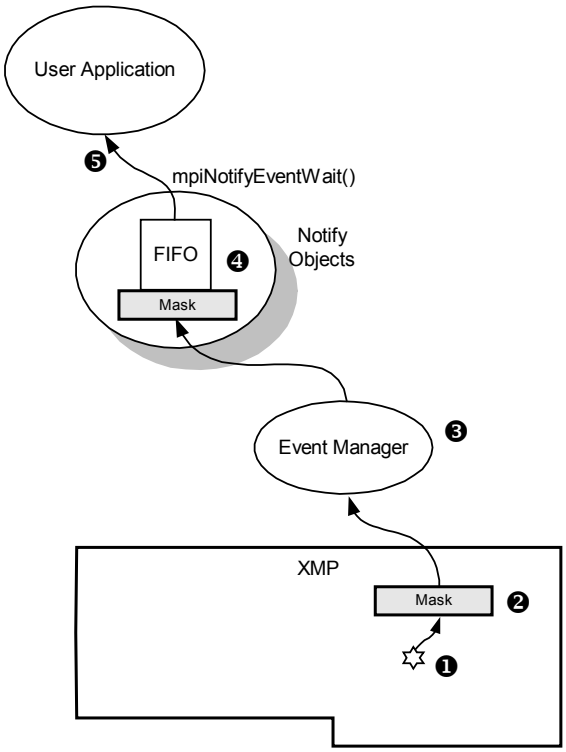


Figure 9: CAN Events

Example Applications

Using the I/O

The following application demonstrates how a simple program would access the IO on CANOpen nodes.

```
#include <assert.h>
#include "stdmpi.h"
#include "stdmei.h"

void main(void) {
    MPIControl ControlHandle;
    MEICan CANHandle;
    long Bit;

    /* Create, validate and initalise a handle to the default controller. */
    ControlHandle = mpiControlCreate( MPIControlTypeDEFAULT, NULL );
    assert( ControlHandle != MPIHandleVOID );
    assert( mpiControlValidate( ControlHandle ) == MPIMessageOK );
    assert( mpiControlInit( ControlHandle ) == MPIMessageOK );

    /* Create a handle to the CAN object. */
    CANHandle = meiCanCreate( ControlHandle, 0 );
    assert( CANHandle != MPIHandleVOID );
    assert( meiCanValidate( CANHandle ) == MPIMessageOK );

    /* Read input bit 0 from node 3. */
    assert( meiCanNodeDigitalInputGet( CANHandle, 2, 0, &Bit ) == MPIMessageOK );

    /* Echo bit to output 1 on node 4. */
    assert( meiCanNodeDigitalOutputSet( CANHandle, 4, 1, Bit ) == MPIMessageOK );

    /* Delete the MPI Objects. */
    assert( meiCanDelete( CANHandle ) == MPIMessageOK );
    assert( mpiControlDelete( ControlHandle ) == MPIMessageOK );
}
```

CAN Object Methods

meiCanCommand
meiCanConfigGet
meiCanConfigSet
meiCanCreate
meiCanDelete
meiCanEventNotifyGet
meiCanEventNotifySet
meiCanFirmwareDownload
meiCanFirmwareUpload
meiCanMemory
meiCanMemoryGet
meiCanMemorySet
meiCanNodeAnalogInputGet
meiCanNodeAnalogOutputGet
meiCanNodeAnalogOutputSet
meiCanNodeDigitalInputGet
meiCanNodeDigitalInputsGet
meiCanNodeDigitalOutputGet

meiCanNodeDigitalOutputSet
meiCanNodeDigitalOutputsGet
meiCanNodeDigitalOutputsSet
meiCanNodeInfo
meiCanNodeStatus
meiCanInfo
meiCanStatus
meiCanReset
meiCanValidate

meiCanCommand

Syntax

```
long meiCanCommand(MEICan         can,
                   MEICanCommand* command);
```

Description

This function allows set of basic commands to be performed. The "type" field of the MEICanCommand structure specifies the type of command to perform.

MEICanCommandTypeSDO_READ

This command reads the remote nodes object dictionary using the SDO protocol.

Command data:
- data[0] = Node
- data[1] = Index
- data[2] = SubIndex
- data[3] = Length

Returned data:
- data[0] = Error code
- data[4] = Low Data word
- data[5] = High Data word

MEICanCommandTypeSDO_WRITE

This command writes to a remote nodes object dictionary using the SDO protocol.

Command data:
- data[0] = Node
- data[1] = Index
- data[2] = SubIndex
- data[3] = Length
- data[4] = Low Data word
- data[5] = High Data word

Returned data:
- data[0] = Error code

MEICanCommandTypeCLEAR_STATUS_BITS

Clear selected MEICanStatusBits.

Command data:
- data[0], Bit map of MEICanStatusBits to clear.

Returned data:
- data[0] = Error code

MEICanCommandTypeBUS_START
This puts the CAN bus into operational state if it is Bus off

Command data:
None
Returned data:
data[0] = Error code

MEICanCommandTypeBUS_STOP
This puts the CAN bus into operational state if it is Bus off.

Command data:
None
Returned data:
data[0] = Error code

MEICanCommandTypeNMT_STOP
This puts a node into NMT stopped state.

Command data:
data[0], node to stop, zero is all nodes
Returned data:
data[0] = Error code

MEICanCommandTypeNMT_START
This puts the node into NMT operational state.

Command data:
data[0], node to start, zero is all nodes
Returned data:
data[0] = Error code

Arguments

can	Handle to the CAN object to use.
command	A pointer to a structure which contains the details of the command to be issued, on the functions return it will contain the result of the requested command.

Returns
An MPI error code

meiCanConfigGet

Syntax

```
long meiCanConfigGet(MEICan       can,
                     MEICanConfig* config);
```

Description
This function returns a copy of the current configuration the CAN controller is using.

Arguments

can	Handle to the CAN object to use.
config	A pointer to the CAN configuration structure that will be filled in by this function.

Returns
An MPI error code

See Also
meiCanConfigSet

meiCanConfigSet

Syntax
```
long meiCanConfigSet(MEICan        can,
                     MEICanConfig* config);
```

Description
This function updates the current configuration the CAN controller is using.

Arguments

can	Handle to the CAN object to use.
config	A pointer to the CAN configuration structure containing the new configuration.

Returns
An MPI error code

See Also
meiCanConfigGet

meiCanCreate

Syntax
```
const MEICan meiCanCreate(MPIControl control,
                          long       network);
```

Description
This function creates a CAN object handle that is used subsequently to address the CAN network on this controller. You will need a valid CAN handle to use the MPI's CANOpen functionality.

Arguments

control	Handle to the controller object that contains the CAN object.
network	The number of the CAN network on the specified controller. For most controllers with a single CAN network interface this will be zero. Network numbers are zero based.

Returns
Handle to the CAN object created or MPIHandleVOID.

See Also
MeiCanValidate, meiCanDelete

Example Code
The following code sample shows the creation ad destruction of valid a CAN handle.

```
MPIControl ControlHandle;
MEICan CANHandle;
long Result;

/* Create, validate and initalise a handle to the controller. */
ControlHandle = mpiControlCreate( MPIControlTypeDEFAULT, NULL );
Result = mpiControlValidate( ControlHandle );
assert( Result == MPIMessageOK );

Result = mpiControlInit( ControlHandle );
assert( Result == MPIMessageOK );
```

```
/* Create and validate a handle to the CAN object. */
CANHandle = meiCanCreate( ControlHandle, 0 );
Result = meiCanValidate( CANHandle );
assert( Result == MPIMessageOK );

/* Use the CAN object here */

/* Delete the CAN and Controller objects */
Result = meiCanDelete( CANHandle );
assert( Result == MPIMessageOK );

Result = mpiControlDelete( ControlHandle );
assert( Result == MPIMessageOK );
```

meiCanDelete

Syntax

```
long meiCanDelete(MEICan can);
```

Description

This function deletes the specified CAN object.

Arguments

can	Handle to the CAN object to delete.

Returns

An MPI error code

See Also

meiCanCreate, meiCanValidate

Example Code

See meiCanCreate for an example of how to use meiCanDelete.

meiCanEventNotifyGet

Syntax

```
long meiCanEventNotifyGet(MEICan           can,
                          MEICanEventMask* eventMask);
```

Description

This function gets the current CAN event mask.

Arguments

can	Handle to the CAN object to use.
eventMask	A pointer to the CAN event mask that will be filled in by this function.

Returns

An MPI error code

See Also

meiCanNotifySet

meiCanEventNotifySet

Syntax

```
long meiCanEventNotifySet(MEICan           can,
                          MEICanEventMask* eventMask);
```

Description

This function updates the current CAN event mask.

Arguments

can	Handle to the CAN object to use.
eventMask	A pointer to the new CAN event mask.

Returns

An MPI error code

See Also

meiCanNotifyGet

meiCanFirmwareDownload

Syntax

```
long meiCanFirmwareDownload(MEICan         can,
                            char*          filename,
                            MEICanCallback callback);
```

Description

This function allows the user to upgrade the CAN controller's firmware.

This operation will take some time, probably between 10 and 30 seconds, to perform the download process hence the callback function is provided to allow the current status of the download operation to be reported to the calling application and also to allow the calling application to abort the download if required. The callback function passes to the calling application the progress of the download process and the calling applications normally returns 0 unless it wants to abort the upgrade in which case it returns 1.

Arguments

can	Handle to the CAN object to use.
filename	The filename of the CAN controller firmware. A .out file.
callback	Pointer to the call back function. Pass an address of zero if you do not have a callback function.

Returns

An MPI error code

See Also

meiCanFirmwareErase, meiCanFirmwareUpload

meiCanFirmwareErase

Syntax

```
long meiCanFirmwareErase(MEICan can);
```

Description

This function allows the user to erase the CAN controllers firmware.

Arguments

None

Returns

An MPI error code

See Also

meiCanFirmwareDownload, meiCanFirmwareUpload

meiCanFirmwareUpload

Syntax

```
long meiCanFirmwareUpload(MEICan         can,
                          char*          filename,
                          MEICanCallback callback);
```

Description

This function allows the user to get a copy of the current CAN controller's firmware.

This operation will take some time, probably between 10 and 30 seconds, to perform the upload process hence the callback function is provided to allow the current status of the upload operation to be reported to the calling application and also to allow the calling application to abort the upgrade (if required). The callback function passes to the calling application the progress of the upgrade process and the calling applications normally returns 0 unless it wants to abort the upgrade in which case it returns 1.

Arguments

can	Handle to the CAN object to use.
filename	The filename of the CAN controller firmware. A .out file.
callback	Pointer to the call back function. Pass an address of zero if you do not have a callback function.

Returns

An MPI error code

See Also

meiCanFirmwareErase, meiCanFirmwareDownload

meiCanFlashConfigGet

Syntax

```
long meiCanFlashConfigGet(MEICan        can,
                          void*         flash,
                          MEICanConfig* config);
```

Description

This function returns a copy of the current flash configuration the CAN controller is using.

Arguments

can	Handle to the CAN object to use.
flash	Normally NULL
config	A pointer to the CAN configuration structure that will be filled in by this function.

Returns

An MPI error code

See Also

meiCanFlashConfigSet

meiCanFlashConfigSet

Syntax

```
long meiCanFlashConfigSet(MEICan        can,
                          void*         flash,
                          MEICanConfig* config);
```

Description

This function updates the current flash configuration the CAN sun-system is using.

Arguments

can	Handle to the CAN object to use.
flash	Normally NULL
config	A pointer to the CAN configuration structure containing the new configuration.

Returns

An MPI error code

See Also

meiCanFlashConfigGet

meiCanFlashNodeConfigGet

Syntax

```
long meiCanFlashNodeConfigGet(MEICan            can,
                              void*             flash,
                              long              node,
                              MEICanNodeConfig* nodeConfig);
```

Description

This function returns a copy of the current flash configuration the CAN controller is using.

Arguments

can	Handle to the CAN object to use.
flash	Normally NULL
node	The node number of the CANOpen node.
nodeConfig	A pointer to the CAN node configuration structure that will be filled in by this function.

Returns

An MPI error code

See Also

meiCanFlashNodeConfigSet

meiCanFlashNodeConfigSet

Syntax

```
long meiCanFlashNodeConfigSet(MEICan            can,
                              void*             flash,
                              long              node,
                              MEICanNodeConfig* nodeConfig);
```

Description

This function updates the current flash configuration for the node.

Arguments

can	Handle to the CAN object to use.
flash	Normally NULL
node	The node number of the CANOpen node.
nodeConfig	A pointer to the CAN node configuration structure containing the new configuration.

Returns

An MPI error code

See Also

meiCanFlashNodeConfigGet

meiCanMemory

Syntax

```
long meiCanMemory(MEICan  can,
                  void**  memory);
```

Description

This function returns a pointer to base of the CAN processors DPR. This function is not generally used and is provided for implementing advanced features of the MPI.

Arguments

can	Handle to the CAN object to use.
memory	A pointer to the base of the CAN processors DPR.

Returns

An MPI error code

See Also

meiCanMemoryGet, meiCanMemorySet

meiCanMemoryGet

Syntax

```
long meiCanMemoryGet(MEICan  can,
                     void*   dst,
                     void*   src,
                     long    count);
```

Description

This function copies the specified number of bytes from controllers' memory to the applications memory. This function is not generally used and is provided for implementing advanced features of the MPI.

Arguments

can	Handle to the CAN object to use.
dst	The base address of the destination.
src	The base address of the source.
count	The number of bytes to copy.

Returns

An MPI error code

See Also

meiCanMemory, meiCanMemorySet

meiCanMemorySet

Syntax

```
long meiCanMemorySet(MEICan  can,
                     void    *dst,
                     void    *src,
                     long    count);
```

Description

This function copies the specified number of bytes from the applications memory to the controllers' memory. This function is not generally used and is provided for implementing advanced features of the MPI.

Arguments

can	Handle to the CAN object to use.
dst	The base address of the destination.
src	The base address of the source.
count	The number of bytes to copy.

Returns

An MPI error code

See Also

meiCanMemory, meiCanMemoryGet

meiCanNodeAnalogInputGet

Syntax

```
long meiCanNodeAnalogInputGet(MEICan  can,
                              long    node,
                              long    index,
                              double* data);
```

Description

This function gets the current analog input from the specified CAN Node. The analog data returned is scaled to between ±1.0.

Arguments

can	Handle to the CAN object to use.
node	The node number of the CANOpen node.
index	The index to the analog input on the node.
data	A pointer to where the current analog input is returned.

Returns

An MPI error code

meiCanNodeAnalogOutputGet

Syntax

```
long meiCanNodeAnalogOutputGet(MEICan  can,
                               long    node,
                               long    index,
                               double* data);
```

Description

This function gets the current analog output from the specified CAN node and channel. The analog data returned is scaled to between ±1.0.

Arguments

can	Handle to the CAN object to use.
node	The node number of the CANOpen node.
index	The index to the analog input on the node.

data A pointer to where the current analog output is returned.

Returns
An MPI error code

See Also
meiCanNodeAnalogOuputSet

meiCanNodeAnalogOutputSet

Syntax
```
long meiCanNodeAnalogOutputSet(MEICan  can,
                               long    node,
                               long    index,
                               double  data);
```

Description
This function sets the current analog output for the specified CAN node and channel. The analog data used is assumed to be between ±1.0.

Arguments

can	Handle to the CAN object to use.
node	The node number of the CANOpen node.
index	The index to the analog input on the node.
data	The new analog value to be output.

Returns
An MPI error code

See Also
meiCanNodeAnalogOuputGet

meiCanNodeDigitalInputGet

Syntax
```
long meiCanNodeDigitalInputGet(MEICan  can,
                               long    node,
                               long    bit,
                               long*   data);
```

Description
This function gets the current state of the digital input bit on the specified CAN node.

Arguments

can	Handle to the CAN object to use.
node	The node number of the CANOpen node.
bit	Which bit on this node.
data	A pointer to where the current digital bit is returned.

Returns
An MPI error code

See Also
meiCanNodeDigitalInputsGet

Example Code
The following code sample shows how to interrogate the current state of a single digital input bit on a controller. The variable **Bit** will contain either one or zero depending on the

electrical signal being applied to the input pin on the CANOpen node. See meiCanCreate on how to create the CANHandle.

```
long Bit;
long Result;
Result = meiCanNodeDigitalInputGet( CANHandle,
                                    3, /*node*/
                                    0, /*bit*/
                                    &Bit );
assert( Result == MPIMessageOK );
```

meiCanNodeDigitalInputsGet

Syntax

```
long meiCanNodeDigitalInputsGet(MEICan           can,
                                long             node,
                                MEICanDigitalIO* data);
```

Description

This function gets the current state of all the digital input bits on the specified CAN node.

Arguments

can	Handle to the CAN object to use.
node	The node number of the CANOpen node.
data	A pointer to where the current digital bits are returned.

Returns

An MPI error code

See Also

meiCanNodeDigitalInputGet

meiCanNodeDigitalOutputGet

Syntax

```
long meiCanNodeDigitalOutputGet(MEICan  can,
                                long    node,
                                long    bit,
                                long*   data);
```

Description

This function gets the current state of the digital output bit on the specified CAN node.

Arguments

can	Handle to the CAN object to use.
node	The node number of the CANOpen node.
bit	Which bit on this node.
data	A pointer to where the current digital bit is returned.

Returns

An MPI error code

See Also

meiCanNodeDigitaOutputSet, meiCanNodeDigitaOutputsGet, meiCanNodeDigitaOutputsSet

meiCanNodeDigitalOutputSet

Syntax

```
long meiCanNodeDigitalOutputSet(MEICan  can,
                                long    node,
                                long    bit,
                                long    data);
```

Description

This function changes the state of the digital output bit on the specified CAN node.

Arguments

can	Handle to the CAN object to use.
node	The node number of the CANOpen node.
bit	Which bit on this node.
data	The new state of the digital bit.

Returns

An MPI error code

See Also

meiCanNodeDigitaOutputGet, meiCanNodeDigitaOutputsGet, meiCanNodeDigitaOutputsSet

meiCanNodeDigitalOutputsGet

Syntax

```
long meiCanNodeDigitalOutputsGet(MEICan           can,
                                 long             node,
                                 MEICanDigitalIO* data);
```

Description

This function gets the current state of all the digital output bits on the specified CAN node.

Arguments

can	Handle to the CAN object to use.
node	The node number of the CANOpen node.
data	A pointer to where the current digital bits are returned.

Returns

An MPI error code

See Also

meiCanNodeDigitaOutputGet, meiCanNodeDigitaOutputSet, meiCanNodeDigitaOutputsSet

meiCanNodeDigitalOutputsSet

Syntax

```
long meiCanNodeDigitalOutputsSet(MEICan           can,
                                 long             node,
                                 MEICanDigitalIO* data);
```

Description

This function changes the current state of all the digital output bits on the specified CAN node.

Arguments

can	Handle to the CAN object to use.

node	The node number of the CANOpen node.
data	The new of the digital bits.

Returns
An MPI error code

See Also
meiCanNodeDigitaOutputGet, meiCanNodeDigitaOutputSet, meiCanNodeDigitaOutputsGet

meiCanNodeInfo

Syntax
```
long meiCanNodeInfo(MEICan           can,
                    long             node,
                    MEICanNodeInfo*  nodeInfo);
```

Description
This function returns the node information for the specified node on the CAN network that was generated when the XMP finished scanning the network.

Arguments

can	Handle to the CAN object to use.
node	The node number of the CANOpen node.
nodeInfo	A pointer to where this function will put the node information.

Returns
An MPI error code

See Also
meiCanNodeStatus, meiCanInfo, meiCanStatus

meiCanNodeStatus

Syntax
```
long meiCanNodeStatus(MEICan             can,
                      long               node,
                      MEICanNodeStatus*  nodeStatus);
```

Description
This function gets the instantaneous state of the specified node on the CAN network.

Arguments

can	Handle to the CAN object to use.
node	The node number of the CANOpen node.
nodeStatus	A pointer to where this function will put the node status.

Returns
An MPI error code

See Also
meiCanNodeInfo, meiCanInfo, meiCanStatus

meiCanStatus

Syntax
```
long meiCanStatus(MEICan         can,
                  MEICanStatus*  status);
```

Description
This function gets the instantaneous state of the local CAN interface to the CAN network.

Arguments

can	Handle to the CAN object to use.
node	The node number of the CANOpen node.
status	A pointer to where this function will put the status.

Returns
An MPI error code

See Also
meiCanInfo, meiCanNodeInfo, meiCanNodeStatus

meiCanValidate

Syntax
```
long meiCanValidate(MEICan can);
```

Description
This function validates the specified CAN handle.

Arguments

can	Handle to the CAN object to use.

Returns
An MPI error code

MPIMessageUNSUPPORTED indicates that the XMP does not have a CANOpen interface fitted.

See Also
meiCanCreate, meiCanDelete

Example Code
See meiCanCreate for an example of how to use meiCanValidate.

meiCanVersion

Syntax
```
long meiCanVersion(MEICan         can,
                   MEICanVersion* version);
```

Description
This function returns the version of the firmware being used by the CAN controller.

Arguments

can	Handle to the CAN object to use.
version	A pointer to where this function will put the version information.

Returns
An MPI error code

CAN Object Data Types

All the data types defined for use with the CAN object are listed here alphabetically.

MEICanBitRate

Syntax

```
typedef enum {
    MEICanBitRate1000K = 0,
    MEICanBitRate800K,
    MEICanBitRate500K,
    MEICanBitRate250K,
    MEICanBitRate125K,
    MEICanBitRate50K,
    MEICanBitRate20K,
    MEICanBitRate10K
} MEICanBitRate;
```

Description

This enumerates all the valid bit rates that the CANOpen interface can use.

These are all the recommended bit rates that the CANOpen standard defines. See the Bit Rate section for further information.

MEICanBusState

Syntax

```
typedef enum {
    MEICanBusStateOFF,
    MEICanBusStatePASSIVE,
    MEICanBusStateOPERATIONAL
} MEICanBusState;
```

Description

This enumerates the bus states that the XMP's CAN interface can take. See the CAN Bus State section.

MEICanCallback

Syntax

```
typedef long (*MEICanCallback)(long section, long maximumSection);
```

Description

This is the definition of a call back function used during the firmware download.

MEICanCommand

Syntax

```
typedef struct MEICanCommand {
    MEICanCommandType type;
    long data[6];
} MEICanCommand;
```

Description

This structure holds the command request and response for a meiCanCommand.

Fields

type	The type of CAN command.
data	Data associated with the command.

MEICanCommandType

Syntax

```
typedef enum {
    MEICanCommandTypeSDO_READ,
    MEICanCommandTypeSDO_WRITE,
    MEICanCommandTypeCLEAR_STATUS_BITS,
    MEICanCommandTypeBUS_START,
    MEICanCommandTypeBUS_STOP,
    MEICanCommandTypeNMT_STOP,
    MEICanCommandTypeNMT_START
} MEICanCommandType;
```

Description

This enumerates the different type of commands that can be used with meiCanCommand.

MEICanConfig

Syntax

```
typedef struct MEICanConfig {
    MEICanBitRate bitRate;
    unsigned long cyclicPeriod;
    unsigned long healthPeriod;
    unsigned long nodeNumber;
    unsigned long inhibitTime;
} MEICanConfig;
```

Description

This is a structure that holds the configuration of the CAN object. The default state for this structure is held in flash in the controller and the user can use the meiCanConfig and meiCanFlashConfig to interrogate and change to what the CAN system is currently using or the default.

Fields

bitRate	The bit rate the CAN bus uses.
cyclicPeriod	The period between sending consecutive SYNC messages. Specified in ms, a value of zero will disable the SYNC messages from being produced.
healthPeriod	The period used for checking the health of nodes. Specified in ms, a value of zero will disable the health checking protocol. For nodes that use the node guarding protocol this the node guarding period and for nodes that use the heartbeating protocol this is the heartbeat consumer time, and heartbeat producers are half this period.
nodeNumber	The node number of the XMP on the CAN network. CANOpen requires the master node has a valid node number to implement the heartbeat protocol.
inhibitTime	The global time used for the node health protocols.

MEICanDigitalIO

Syntax

```
typedef struct MEICanDigitalIO {
    unsigned long data[2];
} MEICanDigitalIO;
```

Description

This structure holds the state of all the digital inputs or outputs on a CANOpen node. Note the maximum number of inputs or outputs a single node supports is 64.

MEICanEventType

Syntax

```
typedef enum {
    MEICanEventTypeBUS_STATE,
    MEICanEventTypeRECEIVE_OVERRUN,
    MEICanEventTypeEMERGENGY,
    MEICanEventTypeNODE_BOOT,
    MEICanEventTypeHEALTH,
    MEICanEventTypeDIGITAL_INPUT,
    MEICanEventTypeANALOG_INPUT
} MEICanEventMask;
```

Description

This enumeration is used to define the different events that the CAN object can generate. See the Handling Events section for more details on events from the CAN object.

MEICanEventMask

Syntax

```
typedef enum {
    MEICanEventMaskBUS_STATE,
    MEICanEventMaskRECEIVE_OVERRUN,
    MEICanEventMaskEMERGENGY,
    MEICanEventMaskNODE_BOOT,
    MEICanEventMaskHEALTH,
    MEICanEventMaskDIGITAL_INPUT,
    MEICanEventMaskANALOG_INPUT
} MEICanEventMask;
```

Description

This enumeration is used to enable or disable the generation of the corresponding events from the XMP to the host application. See the Handling Events section for more details on events from the CAN object.

CANEvent

Syntax

```
typedef struct MEICanEvent {
  MEICanEventType type;
  Long data[5];
} MEICanEvent;
```

Description

This structure hold the information returned with any CAN Event.

Fields

type — An enumeration indicating the type of event being reported.

data[5] — The meaning of these data words are dependant upon the *type* field.

MEICanEventTypeBUS_STATE
The BusState has changed.
Data[0] contains the new bus state.

MEICanEventTypeRECEIVE_OVERRUN
The CAN hardware detected a receive overrun.

MEICanEventTypeEMERGENGY
An emergency message was received from a node.
Data[0] contains the node number.
Data[1 to 4] contains the contents of the emergency message.

MEICanEventTypeNODE_BOOT
A node boot message was received from a node.
Data[0] contains the node number.

MEICanEventTypeHEALTH
The health of a node has changed.
Data[0] contains the node number.
Data[1] contains the new node health.

MEICanEventTypeDIGITAL_INPUT
A digital input event was received from a node.
Data[0] contains the node number.
Data[1 to 4] contains the new input state.

MEICanEventTypeANALOG_INPUT
A analogue input event was received from a node.
Data[0] contains the node number.
Data[1 to 4] contains the new input state.

CANHealthType

Syntax

```
typedef enum {
    MEICanHealthTypeNODE_GUARDING,
    MEICanHealthTypeHEART_BEATING
} MEICanHealthType;
```

Description

This enumeration is used to report the health protocol that the XMP is using with each node.

MEICanNodeConfig

Syntax

```
typedef struct MEICanNodeConfig {
    MEICanTransmissionType volatile DigitalOutTransmissionType;
    MEICanTransmissionType volatile AnalogOutTransmissionType;
    MEICanTransmissionType volatile DigitalInTransmissionType;
    MEICanTransmissionType volatile AnalogInTransmissionType;
} MEICanNodeConfig;
```

Description

The configuration of each node on the CAN bus. The user is able to select which type of communication, event or cyclic, is to be used for the different types of IO data that a node supports. See the Transmission Types section for more details.

MEICanNodeInfo

Syntax

```
typedef struct MEICanNodeInfo {
  MEICanNodeType    type;
  unsigned long     digitalInputCount;
  unsigned long     digitalOutputCount;
  unsigned long     analogInputCount;
  unsigned long     analogOutputCount;
  unsigned long     healthType;
} MEICanNodeInfo;
```

Description

This structure describes how many of the different types of IO are on this node.

Fields

type — An enumeration indicating the type of node found at startup, or MEICanNodeTypeNONE if no node was found.

digitalInputCount, digitalOutputCount, analogInputCount and analogOutputCount — The number of each type of input or output supported by this node.

The CANOpen protocol only allows the number of digital inputs and outputs to be interrogated in multiples of eight, i.e. if a node has one digital output the "digitalOutputCount" will return eight.

healthType — The type of health checking protocol being used with this node.

MEICanNodeStatus

Syntax

```
typedef struct MEICanNodeStatus {
    unsigned long live;
    unsigned long nmtState;
} MEICanNodeStatus;
```

Description

This structure holds the current status of a node.

Fields

live — Set if the node is alive, clear if the node is dead.

nmtState — The current NMT state the node is reporting.

MEICanNodeType

Syntax

```
typedef enum {
    MEICanNodeTypeNONE = 0,
    MEICanNodeTypeIO   = 0x0191
} MEICanNodeType;
```

Description

This enumerates the different types node that the XMP has detected. MEICanNodeTypeNONE is returned if no node is found or an unsupported node type is detected.

MEICanNMTState

Syntax

```
typedef enum {
    MEICanNMTStateBOOT_UP,
    MEICanNMTStateSTOPPED,
    MEICanNMTStateOPERATIONAL,
    MEICanNMTStatePRE_OPERATIONAL,
    MEICanNMTStateUNKNOWN,
} MEICanNMTSTATE;
```

Description

This enumerates the NMT (network management) states a node on a CANOpen network can have. The XMPs CAN controller will automatically put all nodes into the *Operational* state during the initialization of the network.

MEICanStatus

Syntax

```
typedef struct MEICanStatus {
    MEICanBusState  busState;
    long transmitErrorCounter;
    long receiveErrorCounter;
    long messageRate;
    long tick;
    long softwareReceiveOverflow;
    long hardwareReceiveOverflow;
} MEICanStatus;
```

Description

The structure hold the current status of the XMP's CAN object.

Fields

busState	The current bus state of the XMP's CAN interface.
transmitErrorCounter	The current value of the transmit error counter.
receiveErrorCounter	The current state of the receive error counter.
messageRate	The number of messages received and transmitted per second.
tick	This is incremented every 1ms by the CAN firmware.
softwareReceiveOverflow	This bit will be set if software receive buffer has overflowed. This bit can be cleared by using the CLEAR_STATUS_BITS command.
hardwareReceiveOverflow	This bit will be set if the CAN interface hardware has detected an overflow. This bit can be cleared by using the CLEAR_STATUS_BITS command.

MEICanTransmissionType

Syntax

```
typedef enum {
    MEICanTransmissionTypeCYCLIC,
    MEICanTransmissionTypeEVENT
} MEICanTransmissionType;
```

Description

This enumerates the transmission types a node can use. See the Transmission Types section for more details.

MEICanVersion

Syntax

```
typedef struct MEICanVersion {
    long bootloaderVersion;
    long interfaceVersion;
    long firmwareVersion;
} MEICanVersion;
```

Description

This structure holds the version information about the XMP's CAN object.

Fields

bootloaderVersion	The version number of the CAN bootloader.
interfaceVersion	A copy of the interface (DPR memory map) version. i.e. "MEIXmpCanVERSION" from xmpcan.h.

firmwareVersion The CAN firmware version.

Error Messages

In addition to the existing MPI error messages the CAN object returns the following CAN specific error messages.

MEICanMessageFIRMWARE_INVALID

No operational CAN firmware was found on the XMP.
Use Motion Console or canFlash.exe to download a CAN?????.out file to the XMP.

MEICanMessageFIRMWARE_VERSION

The CAN firmware executing on the XMP does not match the version the MPI is using to communicate with the CAN interface.
Use Motion Console or canFlash.exe to download a CAN?????.out file to the XMP.

MEICanMessageNOT_INITALISED

The CAN firmware is the correct version but has not started executing.
execute a mpiControlReset() to attempt to re-start the CAN interface.

MEICanMessageIO_NOT_SUPPORTED

You tried to access IO on a node that does not support this type of IO. For example typing to set a digital output on a node that only has digital inputs.

MEICanMessageFILE_FORMAT_ERROR

The .out file supplied to download to the controller was the wrong format.

MEICanMessageUSER_ABORT

The user aborted a firmware upload or download.

MEICanMessageCOMMAND_PROTOCOL

An error was detected when communicating with the CAN controller on the XMP.

MEICanMessageINTERFACE_NOT_FITTED

A CAN interface was not found on this controller.

MEICanMessageNODE_DEAD

You tried to access a node that is dead.
You will need to check the network cables and power to each node before using mpiControlReset() to reset the network.

MEICanMessageSDO_TIMEOUT

While performing an SDO transaction to a node the response message was not returned within the correct timeout period.

MEICanMessageSDO_ABORT

The user aborted an SDO transaction.

MEICanMessageSDO_PROTOCOL

While performing an SDO transaction the message returned from the node did not conform to the CANOpen protocol.

MEICanMessageTX_OVERFLOW

The XMP tried to transmit a message and an internal buffer had overflowed.

MEICanMessageRTR_TX_OVERFLOW

The XMP tried to transmit a message and an internal buffer had overflowed.

MEICanMessageRX_BUFFER_EMPTY

The XMP was expecting to receive a message and an internal buffer was empty.

MEICanMessageBUS_OFF

The XMP tried to use the CAN network and the bus was in the Off state.
You will need to check the network cables and power to each node before using mpiControlReset() to reset the network.