CAN Objects

Introduction

The CAN object allow the user easy access to the I/O nodes connected to a controller's CANOpen interface.

If a 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 a controller.

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.

Bit Rate | Transmission Types | Bus State | CAN Hardware |
Node Health | Emergency Messages | Handling Events | XMP Overview |

Methods

Create, Delete, Validate Methods

meiCan**Create**	Create Can object
meiCan**Delete**	Delete Can object
meiCan**Validate**	Validate Can object

Configuration and Information Methods

meiCan**ConfigGet**	Get Can's configuration
meiCan**ConfigSet**	Set Can's configuration
meiCan**FlashConfigGet**	Get Can's flash configuration
meiCan**FlashConfigSet**	Set Can's flash configuration
meiCan**Status**	Get status of the CAN controller.
meiCan**Version**	Returns the version information about a controller's CAN system.
meiCan**Command**	Get Can's flash configuration
meiCan**NodeConfigGet**	Return a copy of the current configuration
meiCan**NodeConfigSet**	Update the current configuration that the specified CAN node is using.
meiCan**NodeFlashConfigGet**	Get the flash configuration of the Can node
meiCan**NodeFlashConfigSet**	Set the flash configuration of the Can node
meiCan**NodeStatus**	Get the instantaneous state of the local CAN interface.
meiCan**NodeInfo**	Return the node information after the XMP finishes scanning the network.

I/O Methods

meiCan**NodeAnalogInputGet**	Get current analog input
meiCan**NodeAnalogOutputGet**	Get current analog output
meiCan**NodeAnalogOutputSet**	Set current analog output
meiCan**NodeDigitalInputGet**	Get the current state of the digital input bit.
meiCan**NodeDigitalInputsGet**	Get the current state of ALL the digital input bits.
meiCan**NodeDigitalOutputGet**	Get the current state of the digital output bit.
meiCan**NodeDigitalOutputsGet**	Get the current state of all the digital output bits.
meiCan**NodeDigitalOutputSet**	changes the state of the digital output bit.
meiCan**NodeDigitalOutputsSet**	Changes the current state of all the digital output bits.

Event Methods

meiCan**EventNotifyGet**	Get event mask of events for which host notification has been requested
meiCan**EventNotifySet**	Set event mask of events for which host notification will be requested

Firmware Methods

meiCan**FirmwareDownload**	Downloads firmware to the Can controller
meiCan**FirmwareErase**	Erases firmware on the Can controller
meiCan**FirmwareUpload**	Uploads firmware from the Can controller

Memory Methods

meiCan**Memory**	Get address to Can's memory
meiCan**MemoryGet**	Copy data from Can memory to application memory
meiCan**MemorySet**	Copy data from application memory to Recorder memory

Data Types

MEICan**BitRate**

MEICan**BusState**

MEICan**Callback**

MEICan**Command**

MEICan**CommandType**

MEICan**Config**

MEICan**DigitalIO**

MEICan**HealthType**

MEICan**Message**

MEICan**NodeConfig**

MEICan**NodeInfo**

MEICan**NodeStatus**

MEICan**NodeType**

MEICan**NMTState**

MEICan**Status**

MEICan**TransmissionType**

MEICan**Version**

mpiCanCreate

Declaration

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

Required Header

stdmei.h

Description

CanCreate 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.

control	a 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.

Example Code

```
The following sample code shows the creation and destruction
of a valid 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 );
```

Return Values

handle	Handle to the CAN object created or MPIHandleVOID.
MPIHandleVOID	if the object could not be created

See Also mpiCanDelete | mpiCanValidate

meiCanDelete

Declaration

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

Required Header

stdmei.h

Description

CanDelete deletes the specified CAN object.

can	handle to the CAN object to delete.

Example Code

```
See meiCanCreate for an example of how to use meiCanDelete.
```

Return Values

MPIMessageOK	if *CanDelete* successfully deletes a CAN object and invalidates its handle

See Also

meiCanCreate | meiCanValidate

meiCanValidate

Declaration `long meiCanValidate(MEICan can);`

Required Header stdmei.h

Description **CanValidate** validates the specified CAN handle.

can	handle to the CAN object

Example Code

```
See meiCanCreate for an example of how to use meiCanValidate.
```

Return Values	
MPIMessageOK	if *CanValidate* successfully validates that the XMP is properly fitted for the CANOpen interface.
MPIMessageUNSUPPORTED	indicates that the XMP is not properly fitted for the CANOpen interface.

See Also meiCanNodeInfo | meiCanNodeStatus

meiCanConfigGet

Declaration

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

Required Header

stdmei.h

Description

CanConfigGet returns a copy of the current configuration of the CAN controller.

can	a handle to the CAN object
config	a pointer to the CAN configuration structure that will be filled in by this function..

Return Values

MPIMessageOK	if *CanConfigGet* successfully returns the copy of the current configuration of the CAN controller.

See Also

meiCanConfigSet

meiCanFlashConfigGet

Declaration

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

Required Header

stdmei.h

Description

CanFlashConfigGet returns a copy of the current flash configuration that the CAN controller is using.

can	handle to the CAN object
flash	normally NULL
config	a pointer to the CAN configuration structure that will be filled in by this function.

Return Values

MPIMessageOK	if *CanFlashConfigGet* successfully returns a copy of the current flash configuration that the CAN controller is using.

See Also

meiCanFlashConfigSet

meiCanFlashConfigSet

Declaration

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

Required Header

stdmei.h

Description

CanFlashConfigSet updates the current flash configuration that the CAN sun-system is using.

can	handle to the CAN object
flash	normally NULL
config	a pointer to the CAN configuration structure that will be filled in by this function.

Return Values

MPIMessageOK	if *CanFlashConfigSet* successfully updates the current flash configuration that the CAN sun-system is using.

See Also

meiCanFlashConfigGet

meiCanStatus

Declaration

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

Required Header stdmei.h

Description **CanNodeStatus** gets the instantaneous state of the local CAN interface to the CAN network.

can	handle to the CAN object
node	the node number of the CANOpen node.
status	a pointer to where this function will put the status.

Return Values

MPIMessageOK	if *CanStatus* successfully gets the instantaneous state of the local CAN interface to the CAN network.

See Also meiCanNodeInfo | meiCanNodeStatus

meiCanConfigSet

Declaration

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

Required Header

stdmei.h

Description

CanConfigSet updates the current configuration of the CAN controller.

can	a handle to the CAN object
config	a pointer to the CAN configuration structure containing the new configuration.

Return Values

MPIMessageOK	if *CanConfigSet* successfully updates the current configuration of the CAN controller.

See Also

meiCanConfigGet

meiCanVersion

Declaration

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

Required Header

stdmei.h

Description

CanVersion returns the version of the firmware being used by the CAN controller.

can	handle to the CAN object
version	a pointer to where this function will put the version information.

Return Values

MPIMessageOK	if *CanVersion* successfully returns the version of the firmware being used by the CAN controller.

See Also

meiCanCommand

Declaration

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

Required Header

stdmei.h

Description

CanCommand allows a set of basic commands to be performed. The ***type*** field of the MEICanCommand structure specifies the type of command to perform.

can	a handle to the CAN object
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.

Return Values	
MPIMessageOK	if *CanCommand* successfully performs the set of specified commands.

See Also

MEICanCommand

meiCanNodeConfigGet

Declaration

```
long meiCanNodeConfigGet(MEICan               can,
                         long                 node,
                         MEICanNodeConfig*    nodeConfig);
```

Required Header

stdmei.h

Description

CanNodeConfigGet returns a copy of the current configuration that the specified CAN node is using.

can	a handle to the CAN object
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.

Return Values

MPIMessageOK	if *CanNodeConfigGet* successfully returns the copy of the current configuration of the specified CAN node.

See Also

meiCanNodeConfigSet | meiCanConfigGet | meiCanConfigSet

meiCanNodeConfigSet

Declaration

```
long meiCanNodeConfigSet(MEICan               can,
                         long                 node,
                         MEICanNodeConfig*    nodeConfig);
```

Required Header

stdmei.h

Description

CanNodeConfigSet updates the current configuration that the specified CAN node is using.

can	a handle to the CAN object
node	the node number of the CANOpen node
nodeConfig	a pointer to the CAN node configuration structure containing the new configuration.

Return Values

MPIMessageOK	if *CanNodeConfigSet* successfully updates the current configuration of the specified CAN node.

See Also

meiCanNodeConfigGet | meiCanConfigGet | meiCanConfigSet

meiCanNodeFlashConfigGet

Declaration

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

Required Header

stdmei.h

Description

CanNodeFlashConfigGet returns a copy of the current flash configuration of the CAN controller.

can	a handle to the CAN object
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

Return Values

MPIMessageOK	if *CanNodeFlashConfigGet* successfully returns the copy of the current flash configuration of the CAN controller.

See Also

meiCanNodeFlashConfigSet

meiCanNodeFlashConfigSet

Declaration

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

Required Header

stdmei.h

Description

CanNodeFlashConfigSet updates the current flash configuration for the node.

can	a handle to the CAN object
flash	normally NULL
node	the node number of the CANOpen node
nodeConfig	a pointer to the CAN node configuration structure containing the new configuration.

Return Values

MPIMessageOK	if *CanNodeFlashConfigSet* successfully updates the current flash configuration for the node.

See Also

meiCanNodeFlashConfigGet

meiCanNodeStatus

Declaration

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

Required Header

stdmei.h

Description

CanNodeStatus gets the instantaneous state of the specified node on the CAN network.

can	handle to the CAN object
node	the node number of the CANOpen node.
nodeStatus	a pointer to where this function will put the node status.

Return Values

MPIMessageOK	if *CanNodeStatus* successfully gets the instantaneous state of the specified node on the CAN network.

See Also

meiCanNodeInfo | meiCanStatus

meiCanNodeInfo

Declaration

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

Required Header

stdmei.h

Description

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

can	handle to the CAN object
node	the filename of the CAN controller firmware (*.out file).
nodeInfo	a pointer to where this function will put the node information.

Return Values

MPIMessageOK	if *CanNodeInfo* successfully returns the node information for the specified node.

See Also

meiCanNodeStatus | meiCanStatus

meiCanNodeAnalogInputGet

Declaration

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

Required Header

stdmei.h

Description

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

can	handle to the CAN object
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.

Return Values

MPIMessageOK	if *CanNodeAnalogInputGet* successfully gets the current analog input from the specified CAN Node.

See Also

meiCanNodeAnalogOutputGet

Declaration

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

Required Header

stdmei.h

Description

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

can	handle to the CAN object
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.

Return Values

MPIMessageOK	if *CanNodeAnalogOutputGet* successfully gets the current analog output from the specified CAN Node and channel.

See Also

meiCanNodeAnalogOutputSet

meiCanNodeAnalogOutputSet

Declaration

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

Required Header

stdmei.h

Description

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

can	handle to the CAN object
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.

Return Values

MPIMessageOK	if *CanNodeAnalogOutputSet* successfully sets the current analog output for the specified CAN Node and channel.

See Also

meiCanNodeAnalogOutputGet

meiCanNodeDigitalInputGet

Declaration

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

Required Header

stdmei.h

Description

CanNodeDigitalInputGet gets the current state of the digital input bit on the specified CAN node.

(Not to be confused with meiCanNodeDigitalInputsGet.)

can	handle to the CAN object
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

Example Code

The following sample code 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 );
```

Return Values

MPIMessageOK	if *CanNodeAnalogInputGet* successfully gets the current analog input from the specified CAN Node.

See Also

meiCanCreate

meiCanNodeDigitalInputsGet

Declaration

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

Required Header

stdmei.h

Description

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

(Not to be confused with meiCanNodeDigitalInputGet.)

can	handle to the CAN object
node	the node number of the CANOpen node.
data	a pointer to where the current digital bits are returned.

Return Values

MPIMessageOK	if *CanNodeDigitalInputsGet* successfully gets the current state of all the digital input bits on the specified CAN Node.

See Also

meiCanNodeDigitalInputGet

meiCanNodeDigitalOutputGet

Declaration

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

Required Header

stdmei.h

Description

CanNodeAnalogOutputGet gets the current state of the digital output bit on the specified CAN Node.

(Not to be confused with meiCanNodeDigitalOutputsGet.)

can	handle to the CAN object
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.

Return Values

MPIMessageOK	if *CanNodeDigitalOutputGet* successfully gets the current digital output bit on the specified CAN Node.

See Also

meiCanNodeDigitalOutputSet | meiCanNodeDigitalOutputsGet | meiCanNodeDigitalOutputsSet

meiCanNodeDigitalOutputsGet

Declaration

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

Required Header stdmei.h

Description

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

(Not to be confused with meiCanNodeDigitalOutputsGet.)

can	handle to the CAN object
node	the node number of the CANOpen node.
data	a pointer to where the current digital bit is returned.

Return Values

MPIMessageOK	if *CanNodeDigitalOutputsGet* successfully gets the current digital output bits on the specified CAN node.

See Also meiCanNodeDigitalOutputGet | meiCanNodeDigitalOutputSet | meiCanNodeDigitalOutputsSet

meiCanNodeDigitalOutputSet

Declaration

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

Required Header

stdmei.h

Description

CanNodeDigitalOutputSet changes the state of the digital output bit on the specified CAN Node.

(Not to be confused with meiCanNodeDigitalOutputsSet.)

can	handle to the CAN object
node	the node number of the CANOpen node.
bit	which bit on this node.
data	the new state of the digital bit.

Return Values

MPIMessageOK	if *CanNodeDigitalOutputSet* successfully changes the state of the digital output bit on the specified CAN Node.

See Also

meiCanNodeDigitalOutputGet | meiCanNodeDigitalOutputsGet | meiCanNodeDigitalOutputsSet

meiCanNodeDigitalOutputsSet

Declaration

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

Required Header

stdmei.h

Description

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

(Not to be confused with meiCanNodeDigitalOutputSet.)

can	handle to the CAN object
node	the node number of the CANOpen node.
data	the new data of the digital bits.

Return Values

MPIMessageOK	if *CanNodeDigitalOutputsSet* successfully changes the current state of all the digital output bits on the specified CAN node.

See Also

meiCanNodeDigitalOutputGet | meiCanNodeDigitalOutputSet | meiCanNodeDigitalOutputsGet

meiCanEventNotifyGet

Declaration

```
long meiCanEventNotifyGet(MEICan           can,
                          MPIEventMask     *eventMask,
                          void             *external);
```

Required Header

stdmei.h

Description

CanEventNotifyGet gets the current CAN event mask.

can	handle to the CAN object.
***eventMask**	a pointer to the MPI event mask that will be filled in by this function.
***external**	external points to an implementation specific structure. Since there is currently no implementation specific data, NULL should be used.

Return Values

MPIMessageOK	if *EventNotifyGet* successfully gets the current CAN event mask.

See Also

meiCanNotifySet

meiCanEventNotifySet

Declaration

```
long meiCanEventNotiySet(MEICan          can,
                         MPIEventMask    eventMask,
                         void            *external);
```

Required Header

stdmei.h

Description

CanEventNotifySet updates the current CAN event mask.

can	handle to the CAN object.
eventMask	a pointer to the new MPI event mask that will be filled in by this function.
***external**	external points to an implementation specific structure. Since there is currently no implementation specific data, NULL should be used.

Return Values

MPIMessageOK	if *CanEventNotifySet* successfully sets the current CAN event mask.

See Also

meiCanEventNotifyGet

meiCanFirmwareDownload

Declaration

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

Required Header

stdmei.h

Description

CanFirmwareDownload allows the user to upgrade the CAN controller's firmware.

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

can	handle to the CAN object
filename	the filename of the CAN controller firmware (*.out file).
callback	a pointer to the call back function. (Pass an address of zero if you do not have a callback function.)

Return Values

MPIMessageOK	if *CanFirmwareDownload* successfully uploads the CAN controller's firmware.

See Also

meiCanFirmwareErase | meiCanFirmwareUpload

meiCanFirmwareErase

Declaration	`long meiCanFirmwareErase(MEICan can);`
Required Header	stdmei.h
Description	**CanFirmwareErase** allows the user to erase the CAN controllers firmware.

Return Values	
MPIMessageOK	if *CanFirmwareErase* successfully erases the CAN controller's firmware.

See Also meiCanFirmwareDownload | meiCanFirmwareUpload

meiCanFirmwareUpload

Declaration

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

Required Header

stdmei.h

Description

CanFirmwareUpload allows the user to get a copy of the current CAN controller’s firmware.

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

can	handle to the CAN object
filename	the filename of the CAN controller firmware (*.out file).
callback	a pointer to the call back function. (Pass an address of zero if you do not have a callback function.)

Return Values

MPIMessageOK	if *CanFirmwareUpload* successfully retrieves a copy of the current CAN controller's firmware.

See Also

meiCanFirmwareErase | meiCanFirmwareDownload

meiCanMemory

Declaration

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

Required Header

stdmei.h

Description

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

can	handle to the CAN object
memory	a pointer to the base of the CAN processors DPR.

Return Values

MPIMessageOK	if *CanMemory* successfully returns a pointer to the base of the CAN processors DPR.

See Also

meiCanMemoryGet | meiCanMemorySet

meiCanMemoryGet

Declaration

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

Required Header

stdmei.h

Description

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

can	handle to the CAN object
dst	the base address of the destination
src	the base address of the source
count	the number of bytes to copy

Return Values

MPIMessageOK	if *CanMemoryGet* successfully copies the specified number of bytes to the application's memory.

See Also

meiCanMemory | meiCanMemorySet

meiCanMemorySet

Declaration

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

Required Header

stdmei.h

Description

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

can	handle to the CAN object
dst	the base address of the destination
src	the base address of the source
count	the number of bytes to copy

Return Values

MPIMessageOK	if *CanMemorySet* successfully copies the specified number of bytes to the controller's memory.

See Also

meiCanMemory | meiCanMemoryGet

MEICanBitRate

MEICanBitRate

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

Description

CanBitRate enumerates all the valid bit rates that the CANOpen interface can use. These are the recommended bit rates that the CANOpen standard defines.

For more information see the Bit Rate section.

See Also

MEICanBusState

MEICanBusState

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

Description

CanBusState enumerates the bus states that the controller's CAN interface can take.

To see how the CanBusState is displayed in Motion Console, click here.

See Also

Documentation on CAN Bus State

MEICanCallback

MEICanCallback

```
typedef long (*MEICanCallback)(long percentage);
```

Description

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

See Also

MEICanCommand

MEICanCommand

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

Description

CanCommand holds the command request and response for an meiCanCommand.

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

See Also

meiCanCommand

MEICanCommandType

MEICanCommandType

```
typedef enum {
    MEICanCommandTypeSDO_READ,
    MEICanCommandTypeSDO_WRITE,
    MEICanCommandTypeCLEAR_STATUS_BITS,
    MEICanCommandTypeBUS_START,
    MEICanCommandTypeBUS_STOP,
    MEICanCommandTypeNMT_ENTER_PRE_OPERATIONAL,
    MEICanCommandTypeNMT_START_REMOTE_NODE,
    MEICanCommandTypeNMT_STOP_REMOTE_NODE,
    MEICanCommandTypeNMT_RESET_NODE,
    MEICanCommandTypeNMT_RESET_COMMUNICATION,
} MEICanCommandType;
```

Description

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

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 issues the CANOpen NMT command "Enter Pre-Operational" to a node.

Command data:
data[0] = Node number, (0 broadcasts to all nodes)

Returned data:
data[0] = Error code

MEICanCommandTypeNMT_START_REMOTE_NODE

This issues the CANOpen NMT command "Start Remote Node" to a node.

Command data:
data[0] = Node number, (0 broadcasts to all nodes)

Returned data:
data[0] = Error code

MEICanCommandTypeNMT_STOP_REMOTE_NODE

This issues the CANOpen NMT command "Stop Remote Node" to a node.

Command data:
data[0] = Node number, (0 broadcasts to all nodes)

Returned data:
data[0] = Error code

MEICanCommandTypeNMT_RESET_NODE

This issues the CANOpen NMT command "Reset Node" to a node.

Command data:
data[0] = Node number, (0 broadcasts to all nodes)

Returned data:
data[0] = Error code

MEICanCommandTypeNMT_RESET_COMMUNICATION

This issues the CANOpen NMT command "Reset Communication" to a node.

Command data:
data[0] = Node number, (0 broadcasts to all nodes)

Returned data:
data[0] = Error code

See Also

meiCanCommand

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_ENTER_PRE_OPERATIONAL

MEICanConfig

MEICanConfig

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

Description

CanConfig holds the configuration of the CAN object. The default state for this structure is held in the controller's flash. Use the meiCanConfigGet/Set and meiCanNodeConfigGet/Set to interrogate and change to what the CAN system is currently using or the default.

bitRate	The bit rate the CAN bus uses. See also CAN Bit Rate.
cyclicPeriod	The period (milliseconds) between sending consecutive SYNC messages. A value of zero will disable the SYNC messages from being produced. See also CAN Transmission Types.
healthPeriod	The period (milliseconds) used for checking the health of nodes. A value of zero will disable the health checking protocol. For nodes that use the node guarding protocol, this is the node guarding period. For nodes that use the heartbeating protocol, this is the heartbeat consumer time (the heartbeat producers are half this period). See also CAN Node Health.
nodeNumber	The node number of the controller on the CAN network. CANOpen requires that the master node has a valid node number to implement the heartbeat protocol. See also CAN Node Numbers.
inhibitTime	The global time used for the node health protocols. See also CAN Transmission Types.

See Also

meiCanConfigGet | meiCanConfigSet | meiCanNodeConfigGet | meiCanNodeConfigSet |

MEICanDigitalIO

MEICanDigitalIO

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

Description

CanDigitalIO holds the state of all the digital inputs or outputs on a CANOpen node.

NOTE: the maximum number of inputs or outputs on a single node supports is 64.

data	Data associated with the command.

See Also

MEICanHealthType

MEICanHealthType

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

Description

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

See Also

MEICanMessage

MEICanMessage

```
typedef enum {
    MEICanMessageFIRMWARE_INVALID,
    MEICanMessageFIRMWARE_VERSION,
    MEICanMessageNOT_INITALIZED,
    MEICanMessageIO_NOT_SUPPORTED,
    MEICanMessageFILE_FORMAT_ERROR,
    MEICanMessageUSER_ABORT,
    MEICanMessageCOMMAND_PROTOCOL,
    MPICanMessageINTERFACE_NOT_FOUND,
    MEICanMessageNODE_DEAD,
    MEICanMessageSDO_TIMEOUT,
    MEICanMessageSDO_ABORT,
    MEICanMessageSDO_PROTOCOL,
    MEICanMessageTX_OVERFLOW,
    MEICanMessageRTR_TX_OVERFLOW,
    MEICanMessageRX_BUFFER_EMPTY,
    MEICanMessageBUS_OFF,
    MEICanMessageSIGNATURE_INVALID,
} MEICanMessage;
```

Description

MEICanMessageFIRMWARE_INVALID

The CAN firmware is not valid. This message code is returned by meiCanCreate(...) if the CAN hardware bootloader detects no firmware has been loaded or the firmware signature is not recognized. To correct this problem, download valid firmware with meiCanFirmwareDownload(...).

MEICanMessageFIRMWARE_VERSION

The CAN firmware version does not match the software version. This message code is returned by meiCanCreate(...), meiCanFirmwareDownload(...), or meiCanFirmwareUpload(...) if the CAN firmware version is not compatible with the MPI library. To correct this problem, download the proper firmware version with meiCanFirmwareDownload(...).

MEICanMessageNOT_INITIALIZED

The CAN firmware did not initialize. This message code is returned by meiCanCreate(...) if the controller did not copy the configuration structure from flash to memory after power-on or controller reset. To correct this problem, verify the controller firmware is correct and the controller hardware is operating properly.

MEICanMessageIO_NOT_SUPPORTED

The CAN node does not support the specified I/O. This message code is returned by CAN methods that read/write to a digital or analog input/output that is out of range. To prevent this problem, specify a supported I/O bit.

MEICanMessageFILE_FORMAT_ERROR

The CAN firmware file format has an error. This message code is returned by meiCanFirmwareDownload(...) if the specified file has an error in its internal headers. This indicates a corrupted file. To correct this problem, use the original CAN firmware file or reinstall the software distribution.

MEICanMessageUSER_ABORT

The CAN firmware loading was aborted. This message code is returned by meiCanFirmwareDownload(...) or meiCanFirmwareUpload(...) when the firmware loading is aborted by the user via the callback function. This message code is returned for application notification. It is not an error.

MEICanMessageCOMMAND_PROTOCOL

The CAN command failed due to a protocol error. This message code is returned by CAN methods that do not get a valid response from a CAN node. To correct this problem, check your CAN nodes for proper operation.

MPICanMessageINTERFACE_NOT_FOUND

The CAN interface is not available. This message code is returned by meiCanCreate(...) if the specified controller does not support a CAN network interface. To correct this problem, use a controller that has a CAN interface.

MEICanMessageNODE_DEAD

The CAN node does not respond. This message code is returned by CAN methods that read/write from a CAN node and the node fails the health check. This message code indicates a node hardware or network connection problem. To correct this problem, verify the node operation and network connections.

MEICanMessageSDO_TIMEOUT

The CAN command failed due to a timeout. This message code is returned by CAN methods that do not get a response from a CAN node within the timeout period. To correct this problem, check your CAN nodes for proper operation.

MEICanMessageSDO_ABORT

The CAN command failed due to a user abort. This message code is returned by CAN methods when an SDO transaction is aborted.

MEICanMessageSDO_PROTOCOL

The CAN command failed due to an SDO protocol error. This message code is returned by CAN methods when an SDO transaction fails because the node did not conform to the CANOpen protocol.

MEICanMessageTX_OVERFLOW

The controller's transmit buffer overflowed. This message code is returned by CAN methods that failed to transmit a message due to an internal memory buffer overflow.

MEICanMessageRTR_TX_OVERFLOW

The controller's transmit buffer overflowed. This message code is returned by CAN methods that failed to transmit a message due to an internal memory buffer overflow.

MEICanMessageRX_BUFFER_EMPTY

The controller's receive buffer is empty. This message code is returned by CAN methods that expected to get a response from a CAN node, but the controller's receive buffer was empty.

MEICanMessageBUS_OFF

The CAN network bus is in the off state. This message code is returned by CAN methods that are not able to use the CAN network because the bus is off. To correct this problem, verify the node operation and network connections.

MEICanMessageSIGNATURE_INVALID

When initialising the CAN system, some tests are performed to make sure that the CAN processor is returning a valid signature value. If an unexpected signature is returned, this error message is returned. A probable cause for this error is that the bootloader is invalid. To correct this problem, you will need to return the controller to MEI to fix the bootloader.

See Also

MEICanNodeConfig

MEICanNodeConfig

```
typedef struct MEICanNodeConfig {
    MEICanTransmissionType digitalOutTransmissionType;
    MEICanTransmissionType analogOutTransmissionType;
    MEICanTransmissionType digitalInTransmissionType;
    MEICanTransmissionType analogInTransmissionType;
} MEICanNodeConfig;
```

Description

CanNodeConfig is the configuration of each node on the CAN bus. You can select which type of communication (event or cyclic) is to be used for the different types of IO data that a node supports.

For more information, see the CAN Transmission Types section.

See Also

MEICanNodeInfo

MEICanNodeInfo

```
typedef struct MEICanNodeInfo {
    MEICanNodeType      type;
    unsigned long       digitalInputCount;
    unsigned long       digitalOutputCount;
    unsigned long       analogInputCount;
    unsigned long       analogOutputCount;
    MEICanHealthType    healthType;
    unsigned long       vendorID;
    unsigned long       productCode;
    unsigned long       versionNumber;
    unsigned long       serialNumber;
} MEICanNodeInfo;
```

Description

CanNodeInfo describes how many of the different types of I/O are on this node.

type	An enumeration indicating the type of node found at startup, or MEICanNodeTypeNONE if no node was found.
digitalInputCount	The number of digital inputs supported by this node. The CANOpen protocol only allows the number of digital inputs to be interrogated in multiples of eight, i.e. if a node has two digital inputs then digitalInputCount will return eight. MEI CANOpen SLICE nodes support an extension to the CANOpen protocol that allows the exact number of digital inputs to be returned in this field.
digitalOutputCount	The number of digital outputs supported by this node. The CANOpen protocol only allows the number of digital outputs to be interrogated in multiples of eight, i.e. if a node has two digital outputs then digitalOutputCount will return eight. MEI CANOpen SLICE nodes support an extension to the CANOpen protocol that allows the exact number of digital outputs to be returned in this field.
analogInputCount	The number of analog inputs supported by this node.
analogOutputCount	The number of analog outputs supported by this node.
healthType	The type of health checking protocol being used with this node. Also see CAN Node Health.
vendorId	This is a number read from the node. Vendor ID numbers are unique numbers allocated to each manufacturer of CANOpen nodes. Not all CANOpen nodes support this feature, in which case, these nodes will return zero for this field. MEI CANOpen nodes always return x014Fh.

productCode	This is a number read from the node. The product code is made up of numbers allocated by each manufacturer to uniquely identify their different types of nodes. Not all CANOpen nodes support this feature, in which case, these nodes will return zero for this field. MEI CANOpen SLICE nodes always return x0204h.
versionNumber	This is a number read from the node. The version number identify the version of code running on this CANOpen node. Not all CANOpen nodes support this feature, in which case, these nodes will return zero for this field. MEI CANOpen nodes do support this field.
serialNumber	This is a number read from the node. The serial number uniquely identifies each CANOpen node. Not all CANOpen nodes support this feature, in which case, these nodes will return zero for this field. MEI CANOpen SLICE nodes do support this field and the number is also on the side label of the Network adapter.

See Also

MEICanNodeStatus

MEICanNodeStatus

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

Description

CanNodeStatus holds the current status of a node.

live	Set if the node is alive, clear if the node is dead.
nmtState	The current NMT state that the node is reporting.

See Also

Documentation on CAN Node Health.

MEICanNodeType

MEICanNodeType

```
typedef enum {
    MEICanNodeTypeNONE = 0,
    MEICanNodeTypeIO   = 401
} MEICanNodeType;
```

Description

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

See Also

MEICanNMTState

MEICanNMTState

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

Description

CanNMTState enumerates the NMT (network management) states of a node on a CANOpen network. The XMP's CAN controller will automatically put all nodes into the Operational state during the initialization of the network.

See Also

MEICanStatus

MEICanStatus

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

Description

CanStatus holds the current status of the XMP's CAN object.

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.

See Also

MEICanTransmissionType

MEICanTransmissionType

```
typedef enum {
    MEICanTransmissionTypeCYCLIC = 0,
    MEICanTransmissionTypeEVENT  = 1,
} MEICanTransmissionType;
```

Description

CanTransmissionType enumerates the transmission types a node can use.

For more information, see the CAN Transmission Types section.

See Also

MEICanVersion

MEICanVersion

```
typedef struct MEICanVersion {
    long    bootloaderVersion;
    long    firmwareVersion;
    char    firmwareRevision;
    long    firmwareSubRevision;
} MEICanVersion;
```

Description

CanVersion holds the version information about the XMP's CAN object.

bootloaderVersion	The version number of the CAN bootloader.
firmwareVersion	The CAN firmware version.
firmwareRevsion	The CAN firmware revision.
firmwareSubRevision	The CAN firmware subrevision.

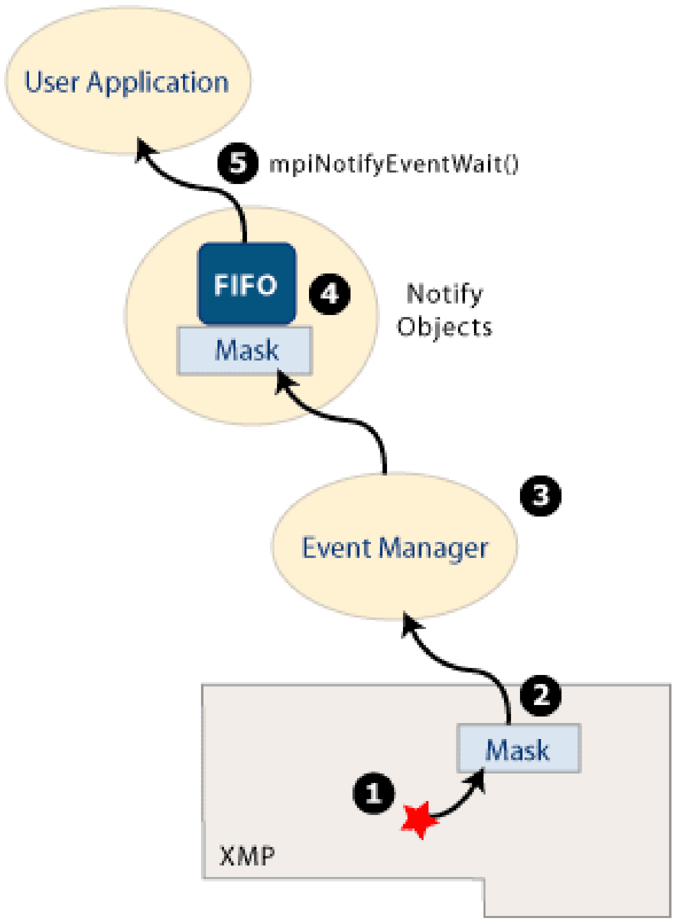
See Also

Handling Events

The CAN interface on the XMP generates many different types of asynchronous events such as:

- 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
- an emergency message is transmitted by a node
- a boot message is transmitted by a node
- a lost message is detected by the XMP CAN firmware

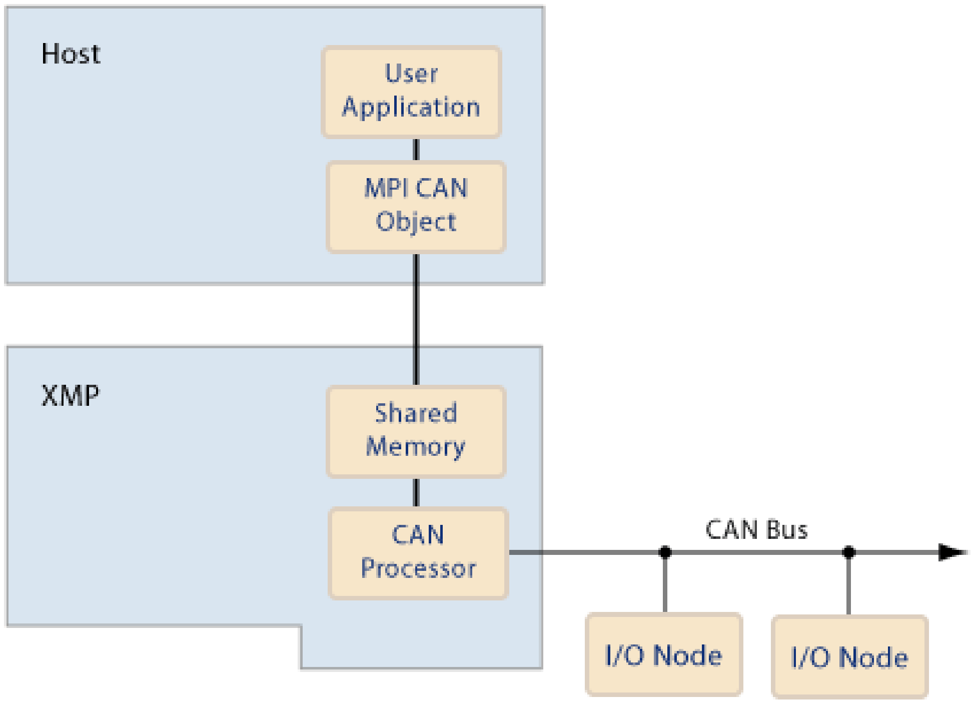
The events above have been appended to the standard MPI event handling scheme in order to provide the user the ability to respond to these events. The diagram below shows an overview of how events are relayed to the user's application.



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 fields of information relevant to the event produced. (ex: the new bus state or node number).

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

CAN Bit Rate

The CANOpen standard defines a set of 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 the table below.

It is recommended that opto-isolated nodes are used on networks with bus lengths longer than 200m.

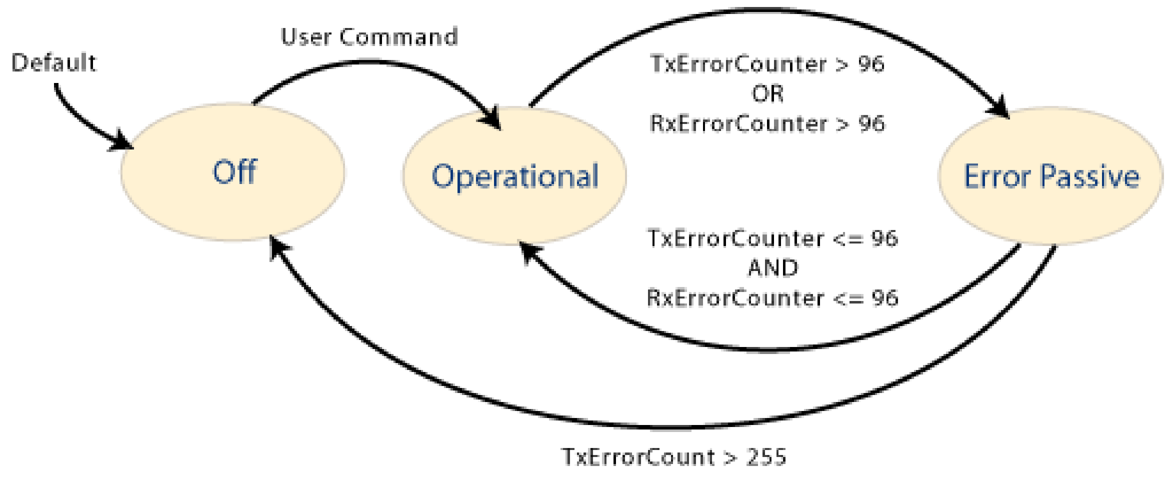
CANOpen Bit Rates

Bit Rate	Max Bus Length (m)	Max Drop Length (m)	Max 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

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:



When a node is in the **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 once the TxErrorCount exceeds 255 error messages. This feature allows nodes that are either malfunctioning or not configured correctly to be isolated for the network, thereby allowing the remaining nodes to successfully communicate.

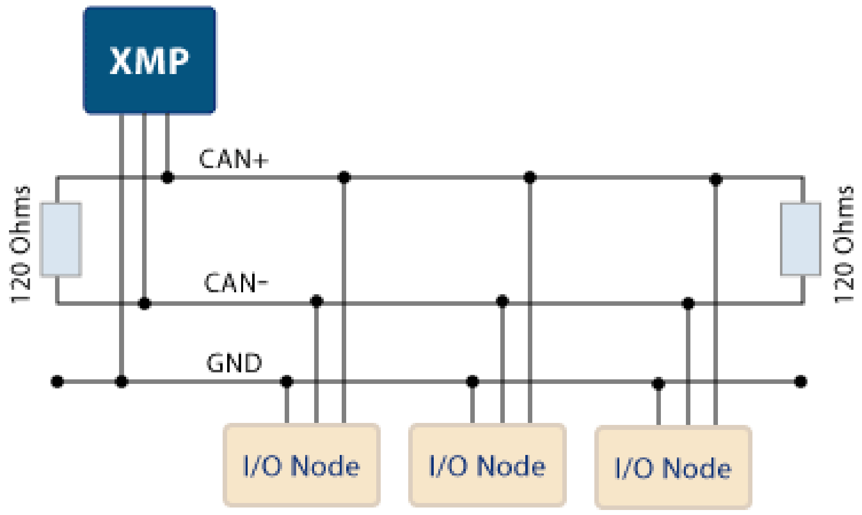
CAN Emergency Messages

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 node 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.

CAN 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 120ohm resistor at both ends of the main bus. If these resistors are not fitted, 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' datasheets for the inclusion of a terminating resistor. The XMP does not have any terminating resistors.



For pinout information, go to the XMP's CAN D-9 connector page.

A CANOpen node either has an opto-isolated or non-isolated interface. The use of optoisolation 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 interfaces, a separate supply of +7 to 24V needs to be provided to power the interface circuitry. The user must also supply an external 24V to the XMP (CAN_V+) if the opto-isolated interface option is being used.

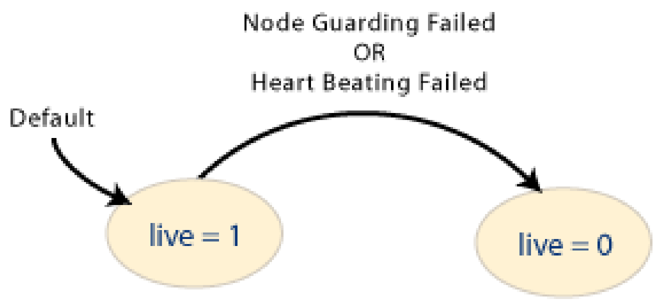
Each node on the network must have a unique node number, in the range of 1 to 127. The node number is commonly set with a bank of DIP switches on each node. If two nodes are given the same node number, network errors are generated and unpredictable problems will be encountered. The node number of the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

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 also CAN Bit Rate.

CAN 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.

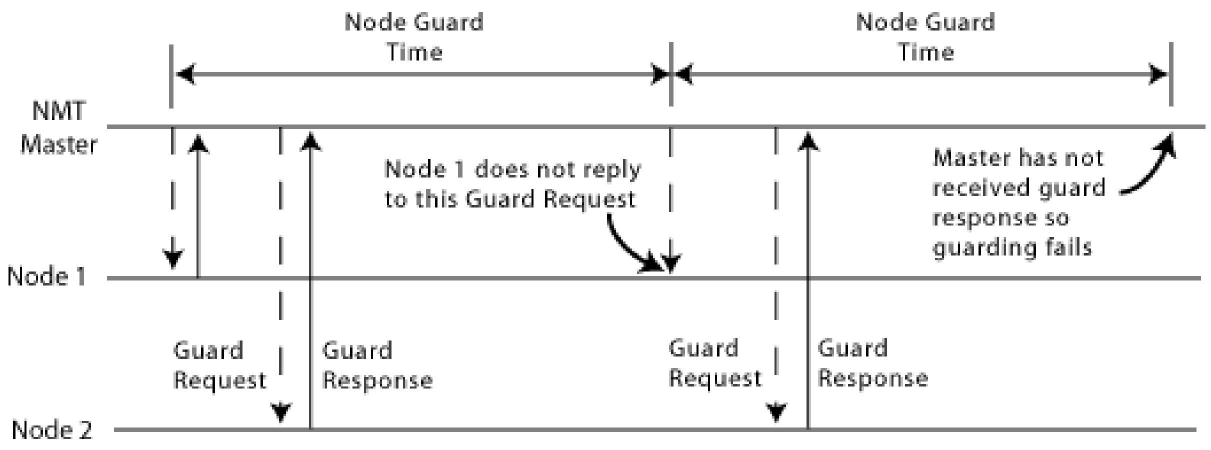
Using these services the XMP can monitor the health of the communications to each of the nodes. The current health of each node is reported in the live field of the MEICANNodeStatus structure.



It is mandatory for a node to either support the node guarding or heart beating protocols, or to support both. The heartbeat protocol has 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 and will automatically detect the protocol that each node supports and then use the most appropriate protocol for the CAN network. The healthType field of the MEICanNodeInfo structure reports the health checking protocol being used with each node.

Node Guarding protocol

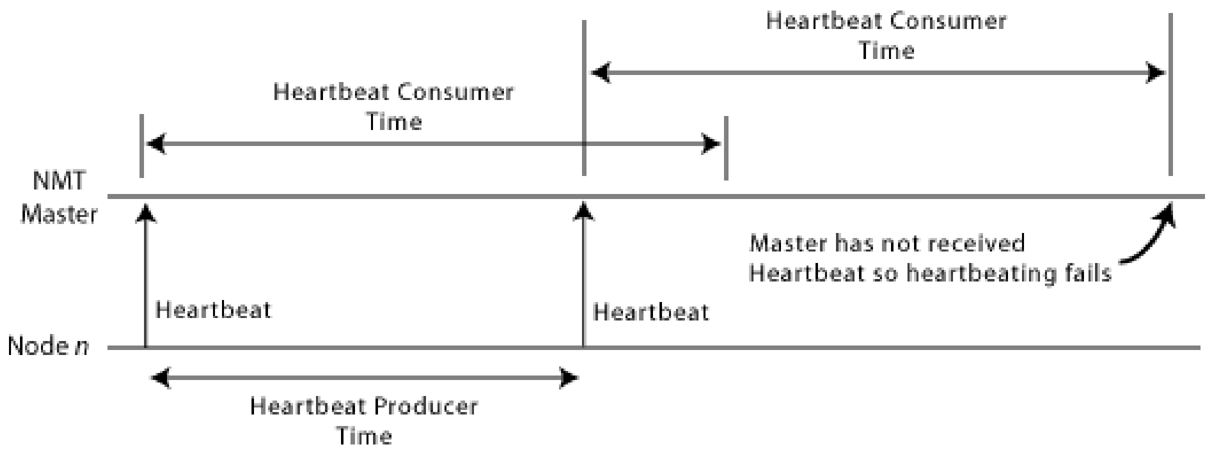
The Node Guarding protocol has the master sending an RTR message to all nodes on the network and checks to see whether a response is received from each of the nodes.



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.



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.

Node Health Times

Protocol Times	Value
Node Guard Time	healthPeriod

Heartbeat Producer Time	healthPeriod / 2
Heatbeat Consumer Time	healthPeriod

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

CAN Node Numbers

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.

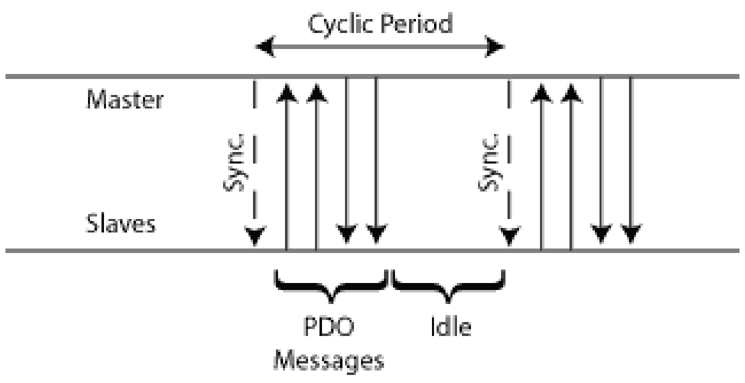
CAN Transmission Types

Introduction

The XMP CANOpen interface uses four messages (serial packets of data on the CAN bus) to pass I/O data between the XMP and an I/O 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 I/O data between the XMP and each of the I/O 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 I/O messages going to and from a node.

Cyclic Transmission

The Cyclic Transmission type, transfers I/O 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. 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.



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 is not the absolute minimum that can be achieved.

Cyclic Period

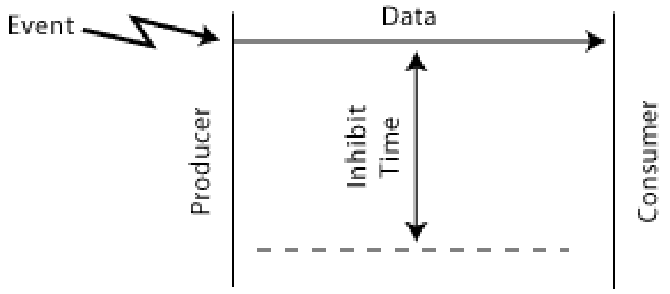
The cyclicPeriod field within the MEICanConfig structure allows the user to specify the period (in milliseconds) that the XMP will use between the successive transmission of synchronization messages. The minimum cyclic period that can be used is dependent upon the chosen bit rate and the number of nodes. 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.

CANOpen Cyclic Period

Bit 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

Event Transmission

The Event Transmission type, only transmits I/O data messages when an “event” occurs on the source node (either the XMP or the I/O node) to change the I/O data. The event that forces the transmission is either a new state of an input that is detected on an I/O node or a new output state that is commanded on the XMP.



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 (ex: 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 and consequently block 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.