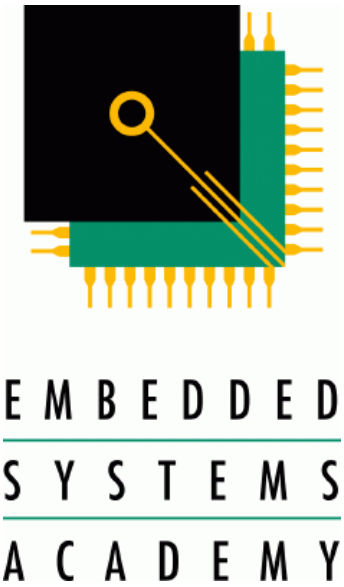

ESAcademy Recommended Practice MicroMessaging

W O R K D R A F T

Rev 0.75 of September 23rd, 2003

References:
[CiADS301] – CANopen Application Layer and Communication Profile, CiA Draft Standard 301, Version 4.02

Table of Content

1 Motivation and Introduction

The goal of MicroMessaging is to create common software interfaces to be used on different embedded networking architectures including, but not limited to UARTs/RS-232/RS-485, I2C, SPI, LIN and CAN.

Figure 1 illustrates the two software interfaces defined: a layer 2 interface (driver) towards the different physical implementation and a layer 7 interface (API) to provide user/application access. The implementation of layers 3 to 7 (the higher-layer protocol) adopt some of the basic features of CANopen [CiADS301].

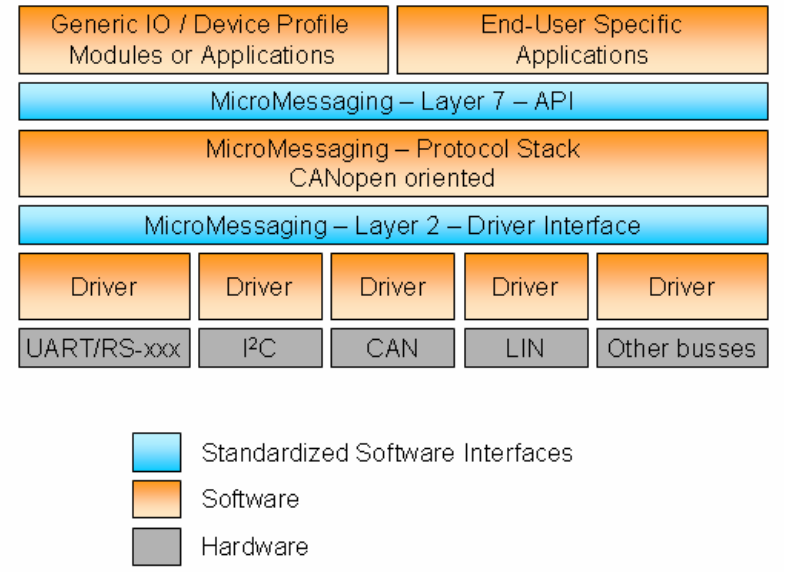

Figure 1: The MicroMessaging Protcol Stack

To reach the goal of a common software interface, two software interfaces are standardized in MicroMessaging:

1. The layer 2 driver interface offers software functions to transmit and receive a message on all the different hardware platforms supported.
2. The layer 7 application programming interface provides a set of application functions that directly allow sharing of process data among the different network nodes.

In order support a wide variety of networking architectures, MicroMessaging supports a flexible usage of the message identifier that is used to identify a message on the network. Although the default is to use a message identifier with 8

bits, MicroMessaging can support network architectures using anywhere from 6 bits (like LIN) to 11 bits (like CAN) for the message identifier.

1.1 Definition of Terms

Checksum

When used on a network architecture that does not provide a build-in checksum, MicroMessaging uses a 4-bit checksum generated by adding all nibbles of a message.

Data Indicator

The data indicator is a byte containing a 4-bit checksum value and the DLC.

DLC – Data Length Code

The data length code indicates how many data bytes are in a single MicroMessaging message. The allowed values are in the range from zero through eight.

Emergency

In severe error situations a MicroMessaging node may produce an emergency message.

Function Code

The message identifier of each MicroMessaging message contains a function code that indicates the function this message has.

Gateway

MicroMessaging networks using different network architectures can be interconnected or connected to a CANopen network using gateways.

Master

A MicroMessaging network requires the presence of a master. Depending on the functionality required, a minimal master might do nothing else but send the master control message that starts the network.

Master Control Message

The master control message is a state-change command to either an individual slave node or a broadcast to all slave nodes. Commands available include start, stop and reset.

Message ID

The message identifier of a single MicroMessaging message contains a function code and a node ID. These values determine from or to which node the message is send and what the content is. Per default, MicroMessaging uses an 8-bit message identifier.

Node-ID

Each MicroMessaging network node must have a unique node ID. The node ID may be in the range of one through 31. The node ID is also used in the message ID.

Object Dictionary

MicroMessaging adopts a minimal Object Dictionary as used by CANopen. It contains the identification and configuration parameters of a slave node.

Poll Message

When used on network architectures without multi-master access (multiple nodes may try to access the network media at the same time), the poll message is used to inform a single slave node that it may now transmit its pending messages.

Process Data Channel

In MicroMessaging each node supports a total of 4 process data channels that are used to communicate process data. Each channel supports up to 8 bytes of process data.

Service Data Channel

In MicroMessaging each node implements two service data channels allowing access to the Object Dictionary: one to receive service data requests and one to transmit service data responses.

1.2 Adopting a minimal CANopen Object Dictionary

MicroMessaging uses the basics of the CANopen Object Dictionary concept [CiADS301] to address identification and configuration parameters in each node. A master can access the Object Dictionary entries of each node using service data request messages (see 2.3.1). To each service data request, the addressed slave must respond with a service data response (see 2.3.2).

A single Object Dictionary entry is addressed using a 16-bit index and an 8-bit subindex value. The data types stored in a single Object Dictionary entry may consist of 1, 2, 3 or 4 bytes.

Table 1 shows the mandatory Object Dictionary entries that every MicroMessaging slave node must implement. In addition, any other CANopen Object Dictionary entry may be implemented, if useful for the particular application.

Index (hex)	Sub-index	Name	Type	Access
1000	0	Device Type	UNSIGNED32	RO
1001	0	Error Register	UNSIGNED8	RO
1017	0	Heartbeat Time	UNSIGNED16	RO/RW
1018	0	Identity Record	UNSIGNED8	RO ('3')
1018	1	Vendor ID	UNSIGNED32	RO
1018	2	Product Code	UNSIGNED32	RO
1018	3	Revision Number	UNSIGNED32	RO

Table 1: Mandatory Object Dictionary Entries

1.2.1 Device Type [1000,00]

With the read-only Device Type entry a node is able to report it general purpose. The low-word of this 4-byte value is set to 0000h to indicate that this device does not follow a standardized CANopen profile. The content of the high-word is application specific and can be used to identify certain types of devices.

1.2.2 Error Register [1001,00]

The read-only Error Register is a 1-byte value that contains eight error flags that can be used to indicate different errors. The Error Register is also part of the optional emergency messages (see 2.2.5).

In MicroMessaging only bit zero is mandatory. If bit zero is set, a generic error occurred. The usage of other bits is optional, but if used should be in accordance to [CiADS301].

1.2.3 Heartbeat Time [1017,00]

The 2-byte heartbeat time specifies the heartbeat frequency of a node in milliseconds. The slave state message (see 2.2.4) is transmitted as the heartbeat

with the frequency specified. Per default, this entry is read-only, however it can also be implemented allowing a read-write access.

1.2.4 Identity Record [1018,00]

This 1-byte, read-only entry should always be three, indicating that three subindex entries are available.

1.2.5 Vendor ID [1018,01]

This 4-byte, read-only entry contains the vendor ID of the manufacturer who produced this device. The vendor ID used is that assigned to the manufacturer by the CiA (www.can-cia.org).

Companies that do not have a CiA assigned vendor ID number should set the low-word of this entry to 0000h.

1.2.6 Product Code [1018,02]

This 4-byte, read-only entry contains a manufacturer specific product identification code. It can be used by manufacturers to distinguish between different MicroMessaging products.

1.2.7 Revision Number [1018,03]

This 4-byte, read-only entry contains a revision number referring to the software and hardware of the device. The high word should be used for a major and the low-word for a minor revision number.

2 The Communication Mechanisms

The message format chosen for MicroMessaging supports a variety of network architectures. Besides CAN, any 8-bit oriented message system is usable for MicroMessaging.

2.1 Message Frame Format

A single MicroMessaging message is shown in Table 2 and consists of a message identifier (default of 8 bits), a data indicator (with a 4-bit checksum and a 4-bit data length counter) and up to 8 data bytes.

The bits in the message identifier are divided into a 3-bit function code stored in the most significant bits and a node ID stored in the least significant bits. The node ID field may use 3, 4 or 5 bits allowing for a maximum of 7, 15 or 31 nodes in one MicroMessaging network.

Name	**Fct. Code**	**Node ID**	**Checksum**	**DLC**	**Data**
Alt. Name	Message ID		Data Indicator		Data Field
Length (bits)	3	3, 4 or 5	4	4	0, 8, 16, 24, 32, 40, 48, 56 or 64

Table 2: MicroMessaging Message Frame

2.1.1 The Message Identifier: Function Code and Node ID

A total of 8 function codes are supported. In conjunction with the node ID, the function code specifies the contents of a message. Table 3 shows the default usage of message identifiers.

Fct. Code	**Node ID**	**Message ID (for 8-bit ID)**	**Message Contents**	**Direction (for Slave)**
0	0	00h	Master Control Message	Rx
0	1 – 31	01h-1Fh	Emergency	Tx
1	0	20h	Poll Message	Rx
1	1 – 31	21h-3Fh	Process Data Channel 1	Default: Tx
2	0	40h	Reserved	n.a.
2	1 – 31	41h-5Fh	Process Data Channel 2	Default: Rx
3	0	60h	Reserved	n.a.
3	1 – 31	61h-7Fh	Process Data Channel 3	Default: Tx
4	0	80h	Reserved	n.a.
4	1 – 31	81h-9Fh	Process Data Channel 4	Default: Rx
5	0	A0h	Reserved	n.a.
5	1 – 31	A1h-BFh	Service Data Request Channel	Rx
6	0	C0h	Reserved	n.a.
6	1 – 31	C1h-DFh	Service Data Response Channel	Tx
7	0	E0h	Master Status	Rx
7	1 – 31	E1h-FFh	Slave Status	Tx

Table 3: Usage of the Message Identifier

The usage of the process data channels (function codes 1 through 4 and nodes ID 1 through 31) may be changed and customized towards the application. The usage of all other message identifiers is fixed and may not be changed.

2.1.2 The Data Indicator: Checksum and DLC

Note: If MicroMessaging is implemented on CAN, the checksum field is not used, as CAN message frames already has a checksum mechanism build-in.

MicroMessaging uses a 4-bit checksum to ensure message integrity. The checksum is build by adding up all nibbles of a MicroMessaging message, including the checksum field itself.

When message identifiers with 6 or 7 bits are used, only the most 2 or 3 significant bits of the message identifier are used as its high nibble.

A message transmitter must ensure that the checksum field is filled with a value that results in the total checksum for the massage to be zero.

A message receiver simply needs to verify that the checksum of the message is zero.

The DLC (Data Length Code) has a value of zero through eight and indicates the number of bytes in the data field.

2.1.3 Data Field Byte Order

MicroMessaging supports numerical data types that consist of multiple bytes. The maximum number of bytes allowed per data type is four. Multiple byte data types are transmitted in Little Endian format. The least significant byte is transmitted first.

2.2 Network Control and Management Messages

The messages defined in this section are used to control and manage the nodes on the network. They include messages to start and stop the network as well as status and emergency messages.

2.2.1 Master Control Message

Function code zero, Node ID zero, broadcast by Master:
The master control message is a message send by the master and allows switching the operation state of single or all slave nodes. All slaves MUST support the reception of this message. The master control message corresponds to the CANopen NMT Master command messages [CiADS301].

Fct. Code	Node ID	DLC	Data 1	Data 2
0	0	2	state	node

Table 4: Master Control Message

Table 4 shows the master control message. Data byte two is used to address a specific node or all nodes. If "node" is set to zero, all nodes are addressed. If set to a number of 1 through 31 the specific node with that node ID is addressed. The possible values of "state" are shown in Table 5.

State	Name	Description
1	Start	The addressed nodes start operating.
2	Stop	The addressed nodes stop operating.
128	Service	The addressed nodes go into service mode.
129	Reset	The addressed nodes reset themselves.

Table 5: State Switch Commands

2.2.2 Poll Message

Function code one, Node ID zero, broadcast by Master or Slave:
The poll message shown in Table 6 is used differently on network architectures that support and those that do not support multi-master access (multiple nodes may write to the network at any time, such as CAN or I^2C).

Fct. Code	Node ID	DLC	Data 1
1	0	1	options

Table 6: Poll Message

On network architectures that do not support multi-master access, the poll message MUST be supported by all slaves as it is used to inform an individual node that it may now transmit all of its pending transmit messages. In this case the poll message affects ALL messages that a node might transmit.

On a network with multi-master access, the poll message ONLY affects the process data messages. Emergency, service and status messages are not affected and can be transmitted at any time. On such networks an "options" value of zero indicates that all nodes are polled, allowing a synchronized data transfer in correspondence to using the SYNC message in CANopen [CiADS301].

Upon receiving its poll message, a node must immediately transmit all of its pending messages. If a node does not start to reply within an application specific time (default is 10ms), the master assumes that the node has nothing to transmit. To avoid length timeouts, it is recommended to use the auto-poll feature described below.

The available "options" of the poll message are shown in Table 7 which lists the individual option bits.

Bit(s)	Name	Description
0..5	Node ID	The node ID of the node to be polled.
6	Priority	If this bit is set, the node should try to minimize its transmissions and only transmit high-priority messages.
7	Auto Poll	If this bit is set, the node will automatically generate the next poll message as soon as it completed all of its transmissions. The poll message generated will have the same contents, but with the node ID incremented by one (node ID 31 is "incremented" to node ID zero)

Table 7: Node Polling Options

The usage of the priority bit is entirely application specific. It allows for more customization when time-critical communication methods must be implemented.

The auto poll bit allows the master to start an automated poll cycle of multiple nodes by just sending one initial poll message. The node currently polled automatically generates the poll message for the next node in line to be polled.

Example: If a network consists of three nodes with the node IDs one through three, the master just transmits a poll message with node ID set to one and the auto poll bit set. Node one will transmit all its messages and then automatically poll node two, which in turn will automatically poll node three once it completed its transmissions. A poll message with the node ID four is an indication for the master that the poll cycle is now completed.

2.2.3 Master State Message

Function code seven, Node ID zero, broadcast by Master:
The master state message as shown in Table 8 is an optional indication to the slaves about the master's operating state. Implementation on both the master and the slave is optional.

Fct. Code	Node ID	DLC	Data 1
7	0	1	state

Table 8: Master State Message

Table 9 shows the possible state values reported by the state message. The values correspond to CANopen operating states [CiADS301].

State	Name	Description
0	Boot	The node just booted-up (after power-on or reset).
4	Stopped	The node stopped all network operations. Only master control messages are used by a node in this state.
5	Running	The node is running.
127	Service	The node is in service mode where no data messages are transmitted or consumed.

Table 9: Operating States

2.2.4 Slave State Message

Function code seven, Node ID 1 through 31, broadcast by Slave:
All slaves MUST support transmitting their state message as shown in Table 10. The transmit frequency is controlled by the heartbeat time described in section 1.2.3.

Fct. Code	Node ID	DLC	Data 1
7	node	1	state

Table 10: Slave State Message

The state values reported are the same as described in Table 9. The node ID field must be set to the node ID of the node transmitting the slave state message.

2.2.5 Emergency Message

Function code zero, Node ID 1 through 31, broadcast by Slave:
The support of emergency messages is optional. If implemented, emergencies give slaves the opportunity to inform the master or other network nodes of serious error conditions the encountered. The content of the emergency message is used as specified by CANopen [CiADS301] and is shown in Table 11.

Fct. Code	Node ID	DLC	Data 1 and 2	Data 3	Data 4 through 8
0	node	8	Error Code	Error Register	Error Field

Table 11: Emergency Message

Selected CANopen Error Codes are shown in , however all CANopen error codes may be used. The Error Register is a copy of the error register described in section 1.2.2. The Error Field is manufacturer specific and may be used for node or application specific error codes.

Code (hex)	Description
00 00	Error reset or no error
10 00	Generic Error
50 00	Device Hardware Error
60 00	Device Software Error
81 00	Communication Error (Layer 2)
82 00	Protocol Error (Layer 7)

Table 12: Selected CANopen Error Codes

If a node recovers from a previous reported emergency it MUST transmit an emergency message with the error code 0000 to inform other nodes of the recovery.

2.3 Service Data Messages

The messages defined in this section are used to detect, maintain and configure the nodes on the network.

2.3.1 Service Data Request Message

Function code five, Node ID 1 through 31, from Master to one Slave:
The message content of a service data request as shown in Table 13 directly corresponds to the CANopen SDO Request [CiADS301] supporting expedited access to an Object Dictionary entry.

Fct. Code	Node ID	DLC	Data 1	Data 2, 3	Data 4	Data 5 to 8
5	node	8	cmd	index	subindex	data

Table 13: Service Data Request Message

The fields 'index" and "subindex" select the Object Dictionary entry to be accessed. In case of a write command "data" contains the data bytes to be written, otherwise the field is unused. The possible values of "cmd" are listed in Table 14.

Cmd	Description
	Read command
	Write command with one data byte
	Write command with two data bytes
	Write command with three data bytes
	Write command with four data bytes

Table 14: Service Data Request Commands

2.3.2 Service Data Response Message

Function code six, Node ID 1 through 31, from one Slave to Master:
The message content of a service data request as shown in Table 15 directly corresponds to the CANopen SDO Request [CiADS301] supporting expedited access to an Object Dictionary entry.

Fct. Code	Node ID	DLC	Data 1	Data 2, 3	Data 4	Data 5 to 8
6	node	8	options	index	subindex	data

Table 15: Service Data Response Message

The fields 'index" and "subindex" confirm the Object Dictionary entry to be accessed. In case of a read command "data" contains the data bytes read, otherwise the field is unused. The possible values of "options" are listed in Table 16.

Options	Description
	Confirmation of write command
	Read response with one data byte
	Read response with two data bytes
	Read response with three data bytes
	Read response with four data bytes
	Object Dictionary access failed

Table 16: Service Data Response Options

In case of a failed service data access, the 4 bytes in the data field contain a service data error code. This error code directly corresponds to the CANopen SDO Abort Codes used in [CiADS301].

Code (hex)	Description

Table 17: Selected Service Access Error Codes

2.4 Process Data Messages

The messages used in this section are entirely used for the exchange of process data. Only nodes that are in the running state can produce or consume these messages.

2.4.1 Process Data Message

Function code 1 through 4, Node ID 1 through 31, configurable:
Process data messages are used to exchange process data between MicroMessaging nodes.

Fct. Code	Node ID	DLC	Data 1 to 8	Default Usage
1	node	1 to 8	data	1st data channel of node, transmit
2	node	1 to 8	data	2nd data channel of node, receive
3	node	1 to 8	data	3rd data channel of node, transmit
4	node	1 to 8	data	4th data channel of node, receive

Table 18: Default Process Data Messages

2.5 Sharing the Network Media

The transmission types (message triggering mechanisms) supported by MicroMessaging depend on the type of network media used. On a network media supporting "multi-master-access" (any node can write to the bus at any time, collisions get resolved automatically – available with CAN and I^2C) nodes may write their messages at any time – collisions get automatically resolved.

If the network media does not support "multi-master-access" (such as a UART interface shared by multiple nodes), all transmissions only happen by polling: The master becomes responsible to poll all the Slaves connected to the network cyclically. Upon being polled, a node may transmit all its messages that are due for transmission. The last message transmitted by a polled node is another poll message. The content of this poll message (which node gets polled next) is configurable.

3 The Software Interfaces

MicroMessaging defines two software interfaces. One at the layer two interfacing to the network architecture used (the driver) and one at the layer seven interfacing to the application (the API),

3.1 The Layer 2 Driver Interface

The functionality of the driver interface was carefully selected to be able to support a wide variety of networking architectures.

3.1.1 MM2.h Definition File

```
/*****************************************************************************
MODULE:    MM2
CONTAINS:  MicroMessaging Layer 2 Driver Interface Definitions
COPYRIGHT: Embedded Systems Academy, Inc. 2003
           See www.MicroMessaging.com
           This software was written in accordance to the guidelines at
           www.esacademy.com/software/softwarestyleguide.pdf
DISCLAIM:  Read and understand our disclaimer before using this code!
           www.esacademy.com/disclaim.htm
LICENSE:   General Public License as specified by GNU
VERSION:   0.75, Pf 23-SEP-03
-----------------------------------------------------------------------------
HISTORY:   0.75, Pf 23-SEP-03, First Published Version
*****************************************************************************/
```

```
/*****************************************************************************
DEFINES TO CONTROL THE OPERATION OF THE DRIVER
*****************************************************************************/

// This defines the number of bits a message identifier has.
// Possible values are 6 (LIN), 8 (default) or 11 (CANopen)
#define NR_OF_MESSAGE_ID_BITS 8

// The following definition controls the network media arbitration
// method used.
//  1: Multi-Master (as available in CAN and I2C), nodes can access the
//     network at any time.
//  2: Master-Polling, nodes are individually polled with a poll message
//     containing the node ID of the node currently polled.
//     POLL_MSG_ID defines the message ID of the poll message used.
#define NET_ARBITRATION 1
#define POLL_MSG_ID 0x20

// The number of transmit and receive data channels is selectable in order
// to be able to optimize the code
#define NR_OF_TX_CHANNELS 2
#define NR_OF_RX_CHANNELS 2

// Defined Service Responses (CANopen SDO responses)
#define DEVICETYPE 0x00030191L // [1000,00]
#define VENDORID   0x01455341L // [1018,01]
#define PRODUCTID  0x00020012L // [1018,02]
#define REVISION   0x00010005L // [1018,03]


/*****************************************************************************
GLOBAL TYPE DEFINITIONS
*****************************************************************************/

// Standard data types
#define BIT   bit
#define BYTE  unsigned char
#define WORD  unsigned int
#define DWORD unsigned long

// Boolean expressions
#define BOOLEAN unsigned char
#define TRUE 0xFF
#define FALSE 0


/*****************************************************************************
GLOBAL FUNCTIONS
*****************************************************************************/

/*****************************************************************************
DOES:    Initializes the driver with the desired bit rate.
         For a complete initialization, MM2_SetMessageFilter must be called
         for every message ID that should be received by the driver.
RETURNS: TRUE, if initialization executed OK.
         FALSE, if initialization failed.
*****************************************************************************/
BYTE MM2_Init
  (
  WORD BitRate // Bitrate in multiple of 100bps
  );
```

```
/*****************************************************************************
DOES:    Sets a single receive filter. The driver only accepts messages
         from the network for which a filter was set.
RETURNS: TRUE, if message ID filter was set successfully.
         FALSE, if message ID filter could not be set.
*****************************************************************************/
BYTE MM2_SetMessageFilter
  (
  BYTE MessageID
  );


/*****************************************************************************
DOES:    Pushes the next message into the transmit queue.
RETURNS: TRUE, if transmit queue accepted the message.
         FALSE, if transmit queue is full.
*****************************************************************************/
BYTE MM2_PushMessage
  (
  BYTE pMM2TxMsg[] // Pointer to MicroMessage
                   // 1: address byte
                   // 2: number of data bytes (max of 8)
                   // 3-10: data bytes
  );


/*****************************************************************************
DOES:    Pulls a message from the receive queue.
RETURNS: TRUE, if message was received.
         FALSE, if receive queue is empty.
*****************************************************************************/
BYTE MM2_PullMessage
  (
  BYTE pMM2RxMsg[] // Pointer to MicroMessage
                   // Caller must provide a buffer of 10 bytes!!!
                   // 1: address byte
                   // 2: number of data bytes (max of 8)
                   // 3-10: data bytes
  );


/*****************************************************************************
DOES:    This function reads a 1 millisecond timer tick. The timer tick
         must be a WORD and must be incremented once per millisecond.
RETURNS: 1 millisecond timer tick
NOTES:   Data consistency must be insured by the driver.
         (On 8-bit systems, disable the timer interrupt incrementing
         the timer tick while executing this function)
         Systems that cannot provide a 1ms tick may consider incrementing
         the timer tick only once every "x" ms, if the increment is by "x".
*****************************************************************************/
WORD MM2_GetTime
  (
  void
  );
```

```
/*****************************************************************************
DOES:    This function compares a WORD timestamp to the internal timer tick
         and returns 1 if the timestamp expired/passed.
RETURNS: 1 if timestamp expired/passed
         0 if timestamp is not yet reached
NOTES:   The maximum timer runtime measurable is 0x8000 (about 32 seconds).
         For the usage in MicroMessaging that is sufficient.
*****************************************************************************/
BYTE MM2_IsTimeExpired
  (
  WORD timestamp // Timestamp to be checked for expiration
  );
/*****************************************************************************
// Recommended implementation for this function (8051 version):
{
WORD time_now;

  EA = 0; // Disable Interrupts
  time_now = gTimCnt;
  EA = 1; // Enable Interrupts
  timestamp++; // To ensure the minimum runtime
  if (time_now > timestamp)
  {
    if ((time_now - timestamp) < 0x8000)
      return 1;
    else
      return 0;
  }
  else
  {
    if ((timestamp - time_now) > 0x8000)
      return 1;
    else
      return 0;
  }
}
*****************************************************************************/


/*****************************************************************************
CALL BACK FUNCTIONS (MUST BE IMPLEMENTED BY APPLICATION)
*****************************************************************************/

/*****************************************************************************
DOES:    The driver calls this call back function if a fatal error occurs.
*****************************************************************************/
void MM2CB_FatalError
  (
  WORD ErrCode // Error Code reported by driver
  );

/*****************************************************************************
DOES:    The MM7 handler calls this if a node reset is requested by
         the network.
*****************************************************************************/
void MM2CB_Reset
  (
  void
  );
```

```
/*****************************************************************************
DOES:    Call-back function that is called by the driver upon receiving
         a MicroMessage. Application can now use MM2_PullMessage to pick
         up the message.
RETURNS: Nothing.
*****************************************************************************/
void MM2CB_MessageReceived
  (
  BYTE MsgID // The message identifier of the MicroMessage received.
  );


/*****************************************************************************
DOES:    Call-back function that is called by the driver upon transmittal
         of a MicroMessage.
RETURNS: Nothing.
*****************************************************************************/
void MM2CB_MessageTransmitted
  (
  BYTE MsgID // The message identifier of the MicroMessage transmitted.
  );
```

3.2 The Layer 7 API

The MicroMessaging layer 7 API provides a simple standardized interface focusing on the process data exchanged. Within a node, the process data exchanged is organized into a process image: an array of bytes.

MicroMessaging uses Process Data Channels for transporting process data. On layer 2 a data channel corresponds to a single message. As a result a single data channel can have up to 8 process data bytes. Each individual node can have up to four process data channels.

An application may place multiple variables into each process data channel. The only limiting rules are:

1. A single variable must consist of 1, 2, 3 or 4 bytes.
2. The total number of bytes in a process data channel is limited to 8.
3. All variables of one process data channel must be located sequentially in the process image (starting at a specified offset, number of bytes as specified by DLC).
4. Numeric multiple-byte variables must be transferred in "Little Endian" format (lowest significant byte comes first).

3.2.1 MM7.h Definition File

```
/*****************************************************************************
MODULE:    MM7
CONTAINS:  MicroMessaging Layer 7 API Definitions
COPYRIGHT: Embedded Systems Academy, Inc. 2003
           See www.MicroMessaging.com
           This software was written in accordance to the guidelines at
           www.esacademy.com/software/softwarestyleguide.pdf
DISCLAIM:  Read and understand our disclaimer before using this code!
           www.esacademy.com/disclaim.htm
LICENSE:   General Public License as specified by GNU
VERSION:   0.75, Pf 23-SEP-03
-----------------------------------------------------------------------------
HISTORY:   0.75, Pf 23-SEP-03, First Published Version
*****************************************************************************/

#include "MM2.h"

// Definition of the process image that holds all process data.
// Default size is 32 bytes (4x8)
BYTE gProcImg[32];


/*****************************************************************************
GLOBAL FUNCTIONS
*****************************************************************************/

/*****************************************************************************
DOES:    Initializes the MicroMessaging protocol stack.
RETURNS: TRUE, if initialization executed OK.
         FALSE, if initialization failed.
*****************************************************************************/
BYTE MM7_Init
  (
  BYTE NodeID,   // MicroMessaging Node ID (1 to 31)
  WORD BitRate,  // Desired bit rate in 100 bits per second
  WORD HeartRate // Default Heartbeat rate in milliseconds
  );


/*****************************************************************************
DOES:    Initializes one of the four data channels available for each node.
RETURNS: TRUE, if initialization executed OK.
         FALSE, if initialization failed.
*****************************************************************************/
BYTE MM7_InitDataChannel
  (
  BYTE Channel,   // Channel number (1-4)
  BYTE Direction, // 0 for receive channel, 1 for transmit channel
  WORD EventTime, // Only used for transmit channels: set to 0 if data
                  // to be transmitted on every poll message or set to
                  // number of milliseconds between transfers
  BYTE MessageID, // 0 for default ID or explicit message ID to be used
  BYTE len,       // Number of data bytes in channel (0-8)
  BYTE offset     // Offset in process image
  );
```

```
/*************************************************************************
DOES:    Operates the MicroMessaging protocol stack. This function must
         be called frequently in the main while(1) background task.
*************************************************************************/
void MM7_ProcessStack
  (
  void
  );
```

4 MicroMessaging Gateway to CANopen

Multiple MicroMessaging networks can be interconnected when CANopen is used as a backbone. Simple gateway tasks running on nodes that have access to both a MicroMessaging network and a CANopen network can provide such an interconnection.

As CANopen can handle up to 127 nodes, multiple MicroMessaging networks can share the same CANopen backbone, if the gateway task supports the use an offset for the node ID used on CANopen. For example it could have nodes 1-31 (01h-1Fh) on its MicroMessaging network, but translate that to nodes 33-63 (21h-3Fh) on the CANopen network.

A gateway task that runs on a microcontroller with access to both CANopen and a MicroMessaging network can be kept very simple if both networks use the pre-defined connection set. This is the default usage scheme for the message identifiers (11-bit on CANopen, 8-bit on MicroMessaging). These schemes divide the message identifiers into a function code (most significant bits) and the node ID (least significant bits). Exchanging messages between MicroMessaging and CANopen only requires a translation of the function code and node ID used on both systems as shown in table Table 19.

Micro Messaging Function Code	MicroMessaging Usage	CANopen Usage	CANopen Function Code
0	Emergency	Emergency	1
1	Process Data Channel 1	TPDO1	3
2	Process Data Channel 2	RPDO1	4
3	Process Data Channel 3	TPDO2	5
4	Process Data Channel 4	RPDO2	6
5	Service Data Response Channel	SDO Response	11
6	Service Data Request Channel	SDO Request	12
7	Status Channel	NMT Control	14

Table 19: Translating Function Codes from MicroMessaging to CANopen