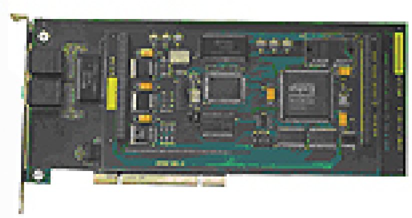
DSPMechatrolink III User's Guide

1 Introduction to DSPMechatrolink III

Product Overview

DSPMechatrolink III is an Isochronous Industrial Network/Controller that functions over the time-tested Ethernet interface. By combining the power of Ethernet with Mechatrolink III protocol that is both simple and reliable, a complete digital solution has been created for networking between motion control elements. The robustness of Ethernet's design is attested to by the fact that it continues to be adapted to new applications, and is constantly being upgraded to provide new capabilities.

YASKAWA Drives



DSPMechatrolink III

When programming with DSPMechatrolink III Network/Controller card, a single Ethernet cable is sufficient to configure and program all devices (such as Yaskawa drives) on the Mechatrolink III network.

Whether the Mechatrolink network is inclusive of a single or multiple devices, DSPMechatrolink as the Isochronous Controller is capable of transmitting the real-time cyclic information through an Ethernet cable in a straight or daisy chained fashion.

Determinism of the Isochronous Ethernet

For high performance motion control applications, such as precise coordination of many motors with less than a microsecond delay between their coordinated commands, Mechatrolink Controller is suited because it comes with an Isochronous Real-Time channel. As indicated by the word "isochronous" in its acronym, Mechatrolink III is used for closed-loop control of a servo system, where the control (both the set-point and feedback) for multiple devices occurs during the same sample period. This sample period can be as strict as 33 microseconds, meaning that the Controller in a Mechatrolink III network issues its command to all devices every 33 microseconds.

Uniqueness of DSPMechatrolink III Controller

Certainly other Ethernet protocols in motion control today operate on a regularly occurring interval basis. A relevant question may be: what is special about Mechatrolink III Network? The guiding factor that sets Mechatrolink III apart from other real-time cyclic protocols is the concept of "jitter". The jitter is defined as a time fluctuation in the start of the interval. For example, in a one-millisecond sample period, if the controller started the next interval 20 nanoseconds after the termination of the previous one, the system could be described as having a jitter of 20 nanoseconds at this point in time.

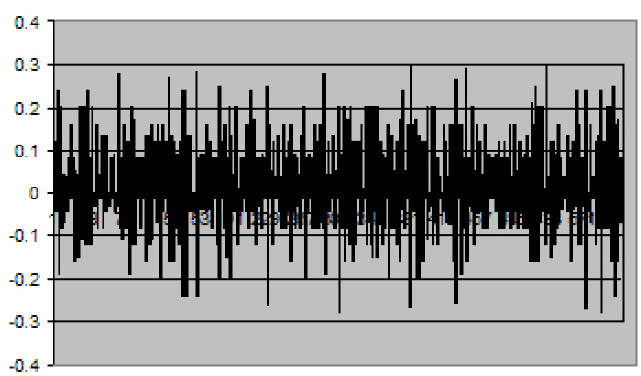


Figure 3: DSPMechatrolink III Jitter in 0.1 Microseconds vs. Sample Time

In the case of Mechatrolink III, both devices and Controller are very concerned with jitter. The threshold for jitter allowed by the Mechatrolink III protocol is defined to be one microsecond. Hence, an entity that wishes to serve as a controller in a Mechatrolink III network must be able to start each cycle very precisely on the aforementioned sample period boundary. The devices in a Mechatrolink III network are designed to be made aware of when a controller is not adhering to the jitter requirement.

The operation of cyclic control at these extremely precise intervals (such as 500 microsecond interval times occurring within fraction of one microsecond of jitter) is what allows for extremely precise coordinated motion control applications to occur across multiple axes.

2 Mechatrolink III Communications

This section describes the specifications of MECHATROLINK-III message communication.

Transmission Frame

This section describes the specifications of the transmission frame that is used for Mechatrolink III. The transmission frame format is shown below.

Preamble	SFD	Destination Address	Source Address	Control Field	Frame Type/ Data Length	Information Field	FCS
56 bit	8 bit	16 bit	16 bit	16 bit	16 bit	32 bytes	32 bit

Preamble, SFD, control field and FCS are used by the DSPMechatrolink communication chip. The destination and source addresses, frame type and data length are set by the access driver. The information field is set by the user application.

Frame Data	Control Layer	Implemented at
Preamble	Data Link Layer	Communication Chip Partly implemented by Access Driver
SFD		
Destination Address		
Source Address		
Control Field	Application Layer	Access Driver
Frame Type		
Data Length		User Application
Information Field		
FCS	Data Link Layer	Communication Chip

Message Communication Frame

The following table shows the frame data used in message communication.

Frame Data	Contents
Destination Address	C1/C2 master station: Sets the station address of a slave station. Slave stations: Sets the station address of the C1/C2 master station.
Source Address	C1/C2 master station: Sets the station address of the local station. Slave stations: Sets the station address of the local station.
Control Field	The communication LSI controls the value.
Frame Type	Fixed at 0Ch (MECHATROLINK message communication). To be set by the access driver.
Data Length	Sets the size of the message to be sent. To be set by the communication chip.
Information Field	Sets the message data. To be set by the user application.

Transmission Sequence

This section describes the transmission sequence of Mechatrolink IIII message communication.

Cyclic Communication Mode

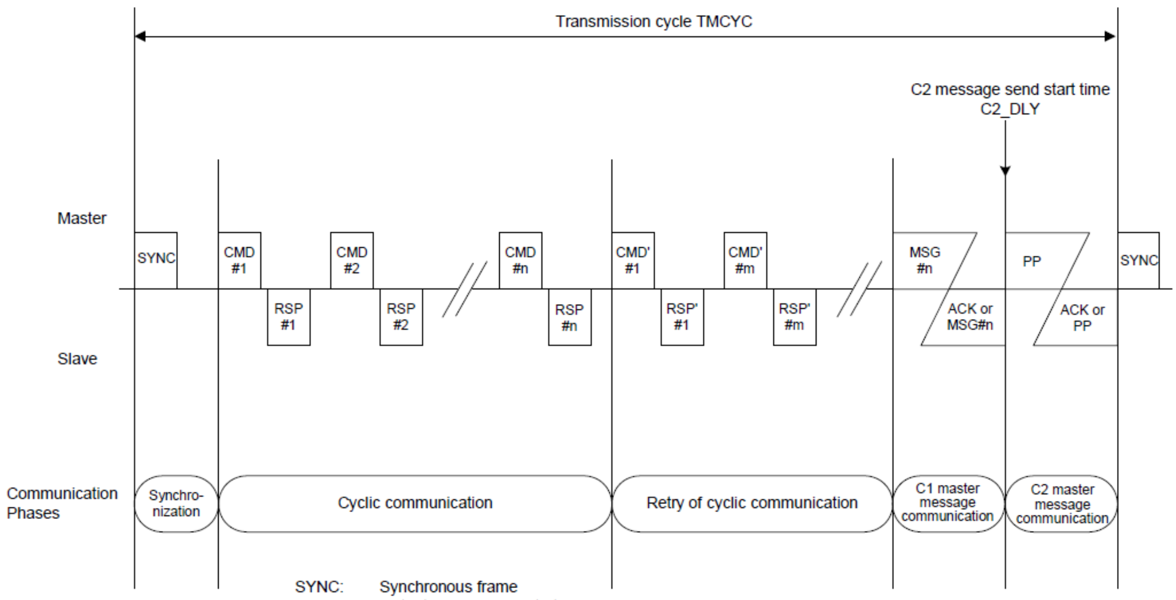
The following describes the transmission sequence of message communication in the cyclic communication mode.

After broadcasting the synchronous frame at the start of the transmission cycle, the C1 master sends the command data and receives the response data once for each slave. The C1 master monitors the response from the slaves and determines the slaves to be retry targets. Slaves from which data reception was abnormal or slaves from which the response data was not received within the response monitoring time are taken as retry targets.

After finishing the exchange of command data and response data for all slaves, the C1 master re-sends the command data to the retry target slaves to receive the response data. After finishing the retry, the C1 master performs C1 message communications if sufficient time is available before the scheduled start of C2 message communications.

If the C1 master completes cyclic communication and C1 message communication before the time to start sending the C2 message, it sends a message token to the C2 master to prompt C2 message communication.

The C2 master performs C2 message communication at the C2 message communication start time or when it receives the message token from the C1 master. C2 message communication continues until the end of the transmission cycle.



SYNC: Synchronous frame
CMD#n: Output (command) data to slave #n
RSP#n: Input (response) data from slave #n
CMD'#m: Retry of sending the output (command) data to slave #m
RSP'#m: Retry of receiving the input (response) data from slave #m
MSG: C1 master message communication
PP: C2 master message communication

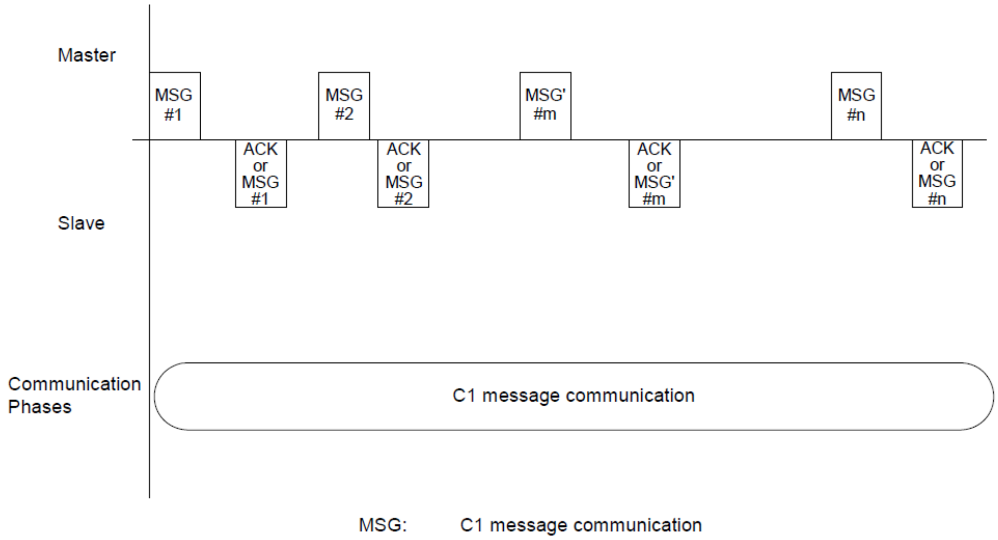
Event Driven Communication Mode

The following describes the transmission sequence of message communication in the event-driven communication mode.

Event-driven communication can be used in a system that does not require synchronized operation (simultaneous operation) of the slaves or when the C1 master collects information for synchronized communication (cyclic communication) from the slaves through C1 message communication.

In the event-driven communication mode, it is possible to execute only the same transmission sequence or C1 message communication as in cyclic communication without fixing the transmission cycle. The same restrictions apply to the data length in event-driven communication as in cyclic communication.

Although all of the C1 master, the C2 master and slaves can participate in event-driven communication, the C2 master can only be used for monitoring and C2 message communication is not possible.

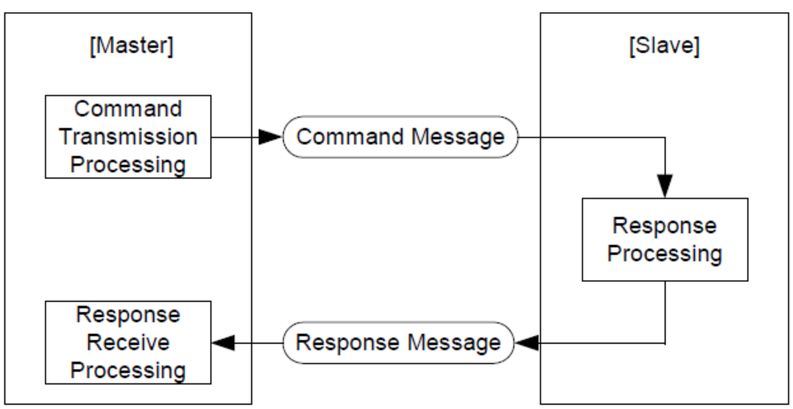


Message communication

This section describes Mechatrolink III message communication.

Communication Method

"Mechatrolink III Message Communication" uses the master/slave communication method (half-duplex system) in which the slave returns the response message in response to the command message sent from the master. In this method, only the master can send the command message (start of communication). The slave executes the function specified in the message and returns the response message.



Message Specifications

Message Format

The message field consists of seven fields for both a command and a response. These fields include the slave address field, the function code field, and the extension field.

Slave Address
Function Code
Extended Address
Reserved (00Hex)
Sub-function Code
Mode/Data Type
Data Count (High)
Data Count (Low)
Data

Slave Address

The slave address (01H to EFH) field. Set the slave address in this field when sending the command message to slaves from the master. The slave reads only the command message addressed to itself. When the slave returns the response message to the master, it sets its own address in the slave address field. The master can recognize the slave that returned the message from the address set in this field.

Function Code

This is a code that shows the function of the MECHATROLINK message and it is fixed as 42H. If the slave returns the response message after executing the specified function, the slave sets the same function code in the response message when the function has been executed normally and it sets "function code + 80H" when it returns an error response message. The master can recognize function code for which the response message was returned from the setting in this field.

Extended Address

This field is only used if extended addresses are used.

Sub-function Code

The master specifies, with a function code, the function that the slave is to execute. The function codes that can be used in MECHATROLINK-III message communication are shown in Function Codes.

Mode/Data Type

Bit7 to bit4: Mode
1H: Specifies volatile memory such as a RAM (normal operation).
2H: Specifies retentive memory such as an E2PROM.
The modes to be supported are specified by the product specifications. In answer to a setting value that is specified as out of range in the product specifications, an error response is returned with the error code 03H (mode/data type error).
bit3 to bit0: Data type
1H: byte type (1-byte)
2H: short type (2-byte)
3H: long type (4-byte)
4H: longlong type (8-byte)

The data types to be supported are specified by the product specifications. In answer to a setting value that is specified as out of range in the product specifications, an error response is returned with the error code 03H (mode/data type error).

Data Count

Specify the data size (big-endian), taking the specified data type as the unit.

Data

The field for the individual function code data. The data length, configuration and meaning are specified for each of the function codes. The data is stored using the big-endian format. For details, refer to the explanation on the message format of the individual function code. A data area of up to 1496 bytes can be used for cyclic communication, and a data area of up to 496 bytes can be used for event-driven communication.

Slave Responses

The messages returned from a slave in response to the command messages sent from the master are classified into the three responses shown below.

Normal Response

When a slave received a command message normally and executed the processing normally, it returns a normal response message.

Error Response

When a slave cannot process the command message although the message was received normally, it returns an error response message. In the error response message, "Function Code + 80H" is set in the function code field and an error code is set in the data field. For the error detection address, the memory address where the slave device first detected an error is set. Whether the data read or written up until an error is detected is to be enabled or disabled is specified by the product specifications.

No Response

A slave does not return a response in the following cases.

- A transmission error (overrun error, framing error, parity error, etc.) is detected in the command message.
- When the slave address specified in the command message does not match with the slave address set for the slave.
- Illegal length of data in the command message.

  In the event of no response, whether the master device retries or sends a communication alarm is specified by the product specifications.

Function Codes

The following table shows the function codes.

Function Code	Function Sub-code	Function
42H	Mechatrolink III Message Function	
	01H	Read Memory
	02H	Write Memory
	03H	Read Memory (non-contiguous)
	04H	Write Memory (non-contiguous)
	06H	Write Memory with Mask
	11H	Read maximum message size
	7FH	User-specific command
	80H-FFH	Not Usable (reserved

* When the action of writing to retentive memory fails, the measures indicated in the product specifications are taken.

Mechatrolink III Message Detail

This section describes details on the MECHATROLINK message functions.
The specification for 32-bit length memory addresses can be specified in the MECHATROLINK message function.
It is possible to access the contents of specified memory addresses in 8-bit, 16-bit, 32-bit, or 64-bit units (as explained in the product specifications).

Read Memory (Sub-function Code: 01H)

Function

This function is used to read the specified data count of the specified memory type from contiguous memory. Data will start to be read from a specified initial address (32-bit length).

The maximum data count that can be read at one time can be calculated from the message size read using the "Read maximum message size" sub-function (sub-function code: 11H).

Message Format

When the Data Type is "byte" (01H)

Data Format (Read Memory)

Byte	Command	Response	
		When Normal	When Abnormal
0	Slave address	Slave address	Slave address
1	Function Code (42H)	Function code (42H)	Function code + 80H (C2H)
2	Extended address (00H)	Extended address	Extended address
3	Reserved (00H)	Reserved (00H)	Reserved (00H)
4	Sub-function code (01H)	Sub-function code (01H)	Sub-function code (01H)
5	Mode/data type (11H)	Mode/data type (11H)	Error code
6	Data count of byte type	Data count of byte type	Data count of byte type
7			
8	Initial address	1th data	Error detection address
9		2nd data	
.		.	
.		nth data	

Example Message

Example of reading three items (long) of memory content from memory address FF00006CH of slave station 2

Data Format (Read Memory)

<table>
<tr><td rowspan="2">Byte</td><td colspan="2" rowspan="2">Command</td><td colspan="2">Response</td><td colspan="2">Response</td></tr>
<tr><td colspan="2">When Normal</td><td colspan="2">When Abnormal</td></tr>
<tr><td>0</td><td>Slave address</td><td>02H</td><td>Slave address</td><td>02H</td><td>Slave address</td><td>02H</td></tr>
<tr><td>1</td><td>Function code</td><td>42H</td><td>Function code</td><td>42H</td><td>Function code + 80H</td><td>C2H</td></tr>
<tr><td>2</td><td>Extended address</td><td>00H</td><td>Extended address</td><td>00H</td><td>Extended address</td><td>00H</td></tr>
<tr><td>3</td><td>Reserved</td><td>00H</td><td>Reserved</td><td>00H</td><td>Reserved</td><td>00H</td></tr>
<tr><td>4</td><td>Sub-function code</td><td>01H</td><td>Sub-function code</td><td>01H</td><td>Sub-function code</td><td>01H</td></tr>
<tr><td>5</td><td>Mode/data type</td><td>13H</td><td>Mode/data type</td><td>13H</td><td>Error code</td><td>02H</td></tr>
<tr><td>6</td><td rowspan="2">Data count of long type</td><td rowspan="2">00H
03H</td><td rowspan="2">Data count of long type</td><td>00H</td><td rowspan="2">Reserved</td><td>00H</td></tr>
<tr><td>7</td><td>03H</td><td>00H</td></tr>
<tr><td>8</td><td rowspan="4">Initial address</td><td>FFH</td><td rowspan="4">1st data</td><td>00H</td><td rowspan="4">Error detection address</td><td>FFH</td></tr>
<tr><td>9</td><td>00H</td><td>00H</td><td>00H</td></tr>
<tr><td>10</td><td>00H</td><td>02H</td><td>00H</td></tr>
<tr><td>11</td><td>6CH</td><td>2BH</td><td>70H</td></tr>
<tr><td>12</td><td colspan="2" rowspan="8"></td><td rowspan="4">2nd data</td><td>00H</td><td colspan="2" rowspan="8"></td></tr>
<tr><td>13</td><td>00H</td></tr>
<tr><td>14</td><td>00H</td></tr>
<tr><td>15</td><td>00H</td></tr>
<tr><td>16</td><td rowspan="4">3rd data</td><td>00H</td></tr>
<tr><td>17</td><td>00H</td></tr>
<tr><td>18</td><td>00H</td></tr>
<tr><td>19</td><td>63H</td></tr>
</table>

3 Data Transfer To and From Mechatrolink III

Memory Access In Dual Port RAM

Data transfer from PC to DSPMechatrolink III (and subsequently to Mechatrolink III nodes)

DSPMechatrolink has two command (transmit) buffers and two response (receive) buffers for each node on the network. The two sets of command buffers are updated alternately so that one set of buffers can be updated by the user's application while the contents of the other set of buffers are being transmitted. Similarly, the response buffers are read out alternately so that one set of buffers can be read while the other set is being filled with received data.

The user's application can either poll DSPMechatrolink's interrupt status to determine when a cyclic transmission cycle should begin, or it can be driven by DSPMechatrolink's IRQ line on the PCI bus.

In either case, once it is time to begin a new Mechatrolink III cycle the user application switches buffers (a single command switches all command and response buffers for all nodes), and for each node which has returned data on the previous bus cycle (determined by a status register) the application reads out the contents of the response buffer for that node, and loads the command buffer for that node with new data.

Each node is allocated 64 bytes of command data and 48 bytes of response data. Command and response data are formatted according to Mechatrolink requirements without any modification by or for DSPMmechatrolink.

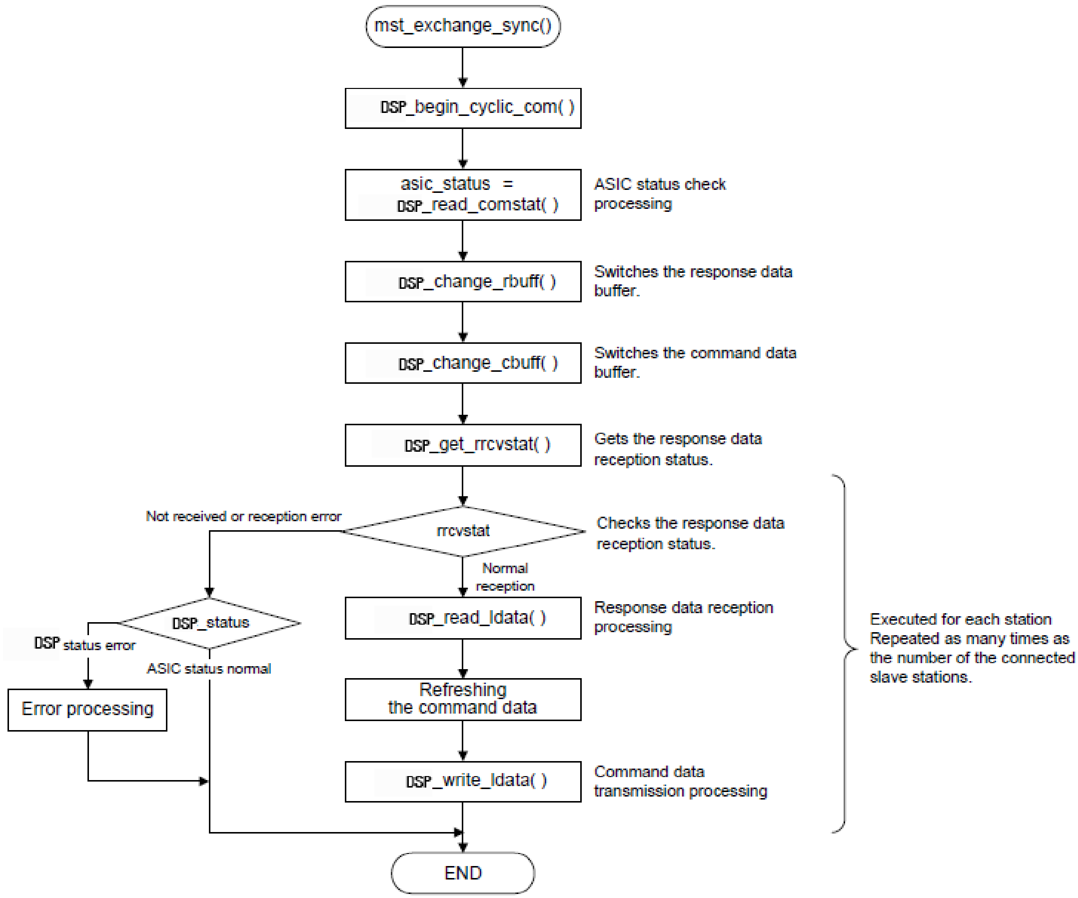
Cyclic Communication Processing

The following describes how cyclic communication processing is executed. In this communication mode, transmission data is written to the DSP (DSPMechatrolink III) and the response data is read every communication cycle. The DSP status is also monitored.

Execute the processing at the start of the communication cycle (INT1) interrupt*.

> * INT1 interrupt occur once every transmission cycle in cyclic communication. To check the interruption factor, use DSP_check_intrp_factor().

Sample code: mst_exchange_sync()



Flowchart of Cyclic Communication Processing

4 Asynchronous and cyclic program examples

Asynchronous Data Exchange For Generic Moves

The following function illustrates an example of sending and receiving data during asynchronous communication.

Where applicable, non-vital parts of the source code have been replaced by comments describing the code, so as not to take away from the relevant code.

What is left is the main algorithm of cyclic processing:

(1) send the asynchronous frame,
(2) receive an asynchronous frame and
(3) perform processing.

```
/*************************************************************************************/
/* exchange_async()                                                                  */
/*                                                                                   */
/* @param      None                                                                  */
/*                                                                                   */
/* Send/recieve data processing (async communication mode).                         */
/*************************************************************************************/
short exchange_async(void)
{
       volatile HOST_IF_REGS *hirp;          // Host I/F Top address
       CHANNEL_INFO*          chbuffp;       // Channel Buffer
       ULONG                  ests;          // Error Status
       USHORT                 st_no;
       ULONG                  ret;           // return code

       // Initialize values, etc.

       // Send async frame
       if (async_sw == ASYNC_SND_SEQ)
       {
              ret = send_frame(chbuffp,
                               DEF_ASYNC_PEER_ADR,
                               DEF_ASYNC_FTYPE,
                               async_sbuff,
                               DEF_ASYNC_DATA_SIZE);

              if (ret == LIB_OK)
              {
                     async_sw = ASYNC_RCV_SEQ;
              }
              else if (ret != SNDING_FRAME)
              {
                     return(ret);
              }
       }

       // Receive async frame
       if (async_sw == ASYNC_RCV_SEQ)
       {
              ret = req_rcv_frame(chbuffp,
                                  &async_rcv_stadr,
```

```
                        &async_rcv_stat,
                        &async_rcv_ftype,
                        async_rbuff,
                        &async_rcv_size,
                        (USHORT)DEF_ASYNC_RCV_TOUT_TIME);
        if (ret == RECEIVED_FRAME)
        {
                if ((async_rcv_stat == ASYNC_RCV_CMP) ||
                    (async_rcv_stat == ASYNC_RCV_TIMOUT))
                {
                        process_async();
                        read_comstat(chbuffp);
                        async_sw = ASYNC_SND_SEQ;
                }
                else
                {
                        // Check communication processor status
                        if ((ests = read_comstat(chbuffp)) != 0)
                        {
                                examine_comstat(chbuffp);
                                return(ERROR_ASIC_STATUS);
                        }
                        return(async_rcv_stat);
                }
                }
                        else if (ret != RCVING_FRAME)
        {
                return(ret);
                }
        }
                return(ret);
}
```

Synchronous Data Exchange For Interpolated Moves

The following function illustrates an example of how sending and receiving data during cyclic communication work.

Where applicable, non-vital parts of the source code have been replaced by comments describing the code, so as not to take away from the relevant code. (For example, error handling has been replaced with comments showing where error handling could be implemented.)

What is left is the main algorithm of cyclic processing:

(1) read in what is fed back to master,
(2) prepare the next outgoing command, and
(3) send the outgoing command.

```
/******************************************************************************/
/*                                                                            */
/* exchange_sync()                                                            */
/*                                                                            */
/* @param      None                                                           */
/*                                                                            */
/* @return     LIB_OK                       Normal end                        */
/*             ERROR_TOUT_CHANG_RBUFF       Change response buffer didn't complete */
/*             ERROR_TOUT_CHANG_CBUFF       Change command buffer didn't complete  */
/*             ERROR_INVALID_STNO           Setting st_no is not exist        */
/*             ERROR_IOMAP_SIZE                                               */
/*                                                                            */
/* Send/receive data processing for sync communication mode.  This will copy */
/* received data from the communication processor's response buffer to rbuff and */
/* copy data to send from sbuff to the communication processor's command buffer. */
/******************************************************************************/
short exchange_sync(void)
{
       CHANNEL_INFO*  chbuffp;              // Channel Buffer
       ULONG          rrcvstat[2];          // Receive status
       USHORT         st_no;                // Counter (iterate through all stations)
       short          ret;                  // Return code

       // Perform some initialization, Start cyclic communication, Check communication
       // processor status, etc.

       // Iterate through all stations (slaves) and see if they have sent us data.
       for (st_no = 1; st_no <= chbuffp->ma_max; st_no++)
       {
              // Check if we have received any data from this station (slave); if
              // not then continue on with checking next station.  We may also
              // check the error status register at this point.

              // If we detect that we have received data, lets get it.  Copy from
              // the DSPMechatrolink III communication processor to the buffer.
              ret = read_ldata(chbuffp, st_no, rbuff[st_no]);
              if (ret != LIB_OK)
              {
                     return (ret);
              }

              process_station_data(st_no, rbuff[st_no]);
       }

       // Set up the data for the outgoing Interpolation Feed command.
       for (st_no = 1; st_no <= chbuffp->ma_max; st_no++)
```

```
	{
		// Every entry is 4 bytes
		sbuff[st_no][0] = IFEED;              // Constant, 0x34
		sbuff[st_no][1] = target_pos[st_no];  // Update its target position
		sbuff[st_no][2] = vel[st_no];         // Update its vff

		// Since we are doing interpolation feed, we shall be interested in
		// monitoring feedback position.  The monitor code for this is 3.  This
		// accounts for one byte.  Then we have two unused bytes, followed by the
		// watchdog byte.  Regarding the watchdog byte, we compose it with the
		// low 4 bits of our counter combined with the high 4 bits of what this
		// particular station sent us for a watchdog value.
		our_watchdog++;
		sbuff[st_no][3] = (((our_watchdog & 0x0f) + (rcvd_watchdog[st_no] & 0xf0))
<< 24) + 3;
	}

	// Iterate through all stations and if the station had sent data to us
	// then we will send data back to it.
	for (st_no = 1; st_no <= chbuffp->ma_max; st_no++)
	{
		// If we received some data from a station then we send new data to it,
		// otherwise we continue on with checking the next station.

		// If we detect that we need to send, lets send.  Copy from our buffer
		// to the communication processor.
		ret = write_ldata(chbuffp, st_no, sbuff[st_no]);
		if (ret != LIB_OK)
		{
			return (ret);
		}
	}

	return (ret);
}
```

DSPMechatrolink III General data Communication

The following functions illustrate examples of synchronous and asynchronous communications.

Also included are system initialization and DSPMechatrolink III memory checking routines that would be performed in start up. Last, this code includes message communication.

```
/********************************************************************************/
/*                                                                              */
/*                  Sample programs for Master With Mechatrolink III            */
/*                                                                              */
/*     The following functions illustrate various possibilities of how Mechatrolink   */
/*     master can operate (e.g. in synchronous and asynchronous communications.)      */
/*     The functions presented here can be called from a main() function to drive     */
/*     the application through the preferred communication style.                     */
/*                                                                              */
/*                                                                              */
/*                                                                              */
/********************************************************************************/
/*                  Transmission Cycle                          : 500 usec     */
#include "gbl.h"
#include "par.h"

// defines
#define HOST_IF_REGS_PTR              0x0f800000     // proc register start address
#define ASYNC_SND_SEQ                 0
#define ASYNC_RCV_SEQ                 1
#define MSG_SND_SEQ                   0
#define MSG_RCV_SEQ                   1

// Error code definitions
#define ERROR_MEASURE_TRANSDLY        (-1)           // A slave could not complete measure
                                                     // transmission delay time
#define ERROR_ASIC_STATUS             (-2)           // Error occurred in ASIC
#define ERROR_SRAM_CHECK              (-3)           // SRAM read/write check error

// globals
// User setting parameter
CHANNEL_INFO                          chbuff;                       // Channel Buffer
USER_PAR                              usr_par;                      // Comm. Parameters
USER_IOMAP                            usr_io_map[DEF_MA_MAX+2];     // IO MAP Parameters

// Buffer
ULONG   sbuff[DEF_MA_MAX+1][(DEF_CD_LEN >> 2)];      // Send Buffer
ULONG   rbuff[DEF_MA_MAX+1][(DEF_RD_LEN >> 2)];      // Receive Buffer
ULONG   c1msg_sbuff[(DEF_C1MSG_SIZE >> 2)];          // Send buffer for C1message comm.
ULONG   c1msg_rbuff[(DEF_C1MSG_SIZE >> 2)];          // Receive buffer for C1message comm.
ULONG   c2msg_sbuff[(DEF_C2MSG_SIZE >> 2)];          // Send buffer for C2message comm.
ULONG   c2msg_rbuff[(DEF_C2MSG_SIZE >> 2)];          // Receive buffer for C2message comm.
ULONG   async_sbuff[(DEF_ASYNC_DATA_SIZE >> 2)];     // Send Buffer for async comm.
ULONG   async_rbuff[(DEF_ASYNC_DATA_SIZE >> 2)];     // Receive Buffer for async comm.

// Work for user setting
USHORT  c1snd_msgsz;                         // Send message data size buffer
USHORT  c1rcv_msgsz;                         // Receive message data size buffer
USHORT  c2snd_msgsz;                         // Send message data size buffer
USHORT  c2rcv_msgsz;                         // Receive message data size buffer
USHORT  async_rcv_stadr;                     // Receive frame source address of async
                                             // comm. buffer
USHORT  async_rcv_ftype;                     // Receive frame type of async comm. buffer
USHORT  async_rcv_size;                      // Receive data size of async comm. buffer
USHORT  async_sw;                            // Async. communication sequence flag
USHORT  msg_sw;                              // Message communication sequence flag

// Status
```

```
USHORT  c1msg_rcv_stat;                      // C1 message receive status
USHORT  c1msg_snd_stat;                      // C1 message send status
USHORT  c2msg_rcv_stat;                      // C2 message receive status
USHORT  c2msg_snd_stat;                      // C2 message send status
USHORT  async_rcv_stat;                      // Async. communication receive status

// Forward declarations
short init(void);                            // Initialize MECHATROLINK communication
                                             // (setup processor)
short exchange_sync(void);                   // Send/Recieve Link data in cyclic
                                             // communication(sync mode)
short exchange_async(void);                  // Send/Recieve Link data with async mode
short exchange_msg(void);                    // Send/Recieve message data
void set_usr_prm(USER_PAR* usr_par,
                 USER_IOMAP* usr_iomapp);    // Set user parameter
short check_ram(ULONG *hif_reg_top,
                USHORT size,
                ULONG chkdata);              // SRAM read/write check

/***************************************************************************************/
/*                                                                                     */
/* mst_init()                                                                          */
/*                                                                                     */
/* Initialize MECHATROLINK communication Setup DSPMechatrolink communication processor)*/
/*                                                                                      */
/***************************************************************************************/
short mst_init(void)
{
       CHANNEL_INFO*          chbuffp;       // Channel Buffer
       USHORT                 st_no;         // Counter
       ULONG                  sti[2];        // Connection status
       short                  ret;           // return code

       // Initialize value
       ret = WAIT_SETUP;
       async_sw = ASYNC_SND_SEQ;
       msg_sw = MSG_SND_SEQ;

       // Get the pointer of Channel Buffer
       chbuffp = &chbuff;

       // Setup ASIC
       while (ret == WAIT_SETUP)
       {
              ret = setup_asic((ULONG *)HOST_IF_REGS_PTR);
              if(ret == LIB_OK)
              {
                     break;
              }
              else if(ret == ERROR_VERIFY_MICRO)
              {
                     return(ret);
              }
       }

       // Check ASIC ready
       while ((ret = chk_asic_ready((ULONG *)HOST_IF_REGS_PTR)) != LIB_OK);

       // Check SRAM area
       ret = check_ram((ULONG *)HOST_IF_REGS_PTR, DEF_SRAM_SIZE, 0x5a5a5a5a);
       if (ret != LIB_OK)
       {
              return(ret);
       }

       ret = check_ram((ULONG *)HOST_IF_REGS_PTR, DEF_SRAM_SIZE, 0xa5a5a5a5);
       if (ret != LIB_OK)
       {
              return(ret);
```

```
}

ret = check_ram((ULONG *)HOST_IF_REGS_PTR, DEF_SRAM_SIZE, 0xffffffff);
if (ret != LIB_OK)
{
        return(ret);
}

ret = check_ram((ULONG *)HOST_IF_REGS_PTR, DEF_SRAM_SIZE, 0);
if (ret != LIB_OK)
{
        return(ret);
}

// Set user parameters
set_usr_prm(&usr_par, usr_io_map);

// Communication processor initialization
ret = initialize( chbuffp, (ULONG *)HOST_IF_REGS_PTR, &usr_par, usr_io_map );
if (ret != LIB_OK)
{
        return(ret);
}

// detect connecting slave
ret = req_detect_slv(chbuffp);
if (ret != LIB_OK)
{
        return(ret);
}

// Check if completed detecting connecting slave
while ((ret = chk_detect_slv_cmp(chbuffp)) != LIB_OK)
{
        if (ret == ERROR_TX_FRM)
        {
                ret = req_detect_slv(chbuffp);
                if (ret != LIB_OK)
                {
                        return(ret);
                }
        }
        else if (ret != WAIT_CMP_DETECT)
        {
                return(ret);
        }
}

for (st_no = 1; st_no <= chbuffp->ma_max; st_no++)
{
        if (read_slvstat(chbuffp, st_no) < STSNUM_WAIT_MEASURE_DLY)
        {
                init_slave(st_no);
        }
}

// Activate user setting parameter
ret = activate_comprm(chbuffp, &usr_par, usr_io_map);
if (ret != LIB_OK)
{
        return(ret);
}

// Measure transmission delay time
ret = req_measure_transdly(chbuffp);
if (ret != LIB_OK)
{
        return(ret);
}
```

```
// Check if completed measuring transmit delay time
while ((ret = chk_transdly_cmp(chbuffp)) != LIB_OK)
{
	if (ret == ERROR_TX_FRM)
	{
		ret = req_measure_transdly(chbuffp);
		if (ret != LIB_OK)
		{
			return(ret);
		}
	}
	else if (ret != MEASURING_TRANSDLY)
	{
		return(ret);
	}
}

get_stistat(chbuffp, sti);
for (st_no = 1; st_no <= chbuffp->ma_max; st_no++)
{
	if (st_no < 32)
	{
		if (((sti[0] >> st_no) & 0x0001) != 0)
		{
			if (read_slvstat(chbuffp, st_no) != STSNUM_WAIT_TMCFRM)
			{
				handle_conn_up(st_no);
			}
		}
	}
	else
	{
		if (((sti[1] >> (st_no - 32)) & 0x0001) != 0)
		{
			if (read_slvstat(chbuffp, st_no) != STSNUM_WAIT_TMCFRM)
			{
				handle_conn_up(st_no);
			}
		}
	}
}

// Check that all slaves completed transmission delay time measurement
for (st_no = 0; st_no < chbuffp->ma_max; st_no++)
{
	ret = read_slvstat(chbuffp, st_no+1);
	if (ret == 0x0000)
	{
		continue;
	}
	else if (ret != STSNUM_WAIT_TMCFRM)
	{
		if (ret == ERROR_INVALID_STNO)
		{
			return(ret);
		}
		else
		{
			handle_slave_not_tx(st_no);
		}
	}
}

// Calculate response monitoring time and interrupt delay time
ret = calc_dlytime(chbuffp, &usr_par, usr_io_map);
if (ret != LIB_OK)
{
	return(ret);
```

```
	}

	// Activate user setting parameter
	ret = activate_comprm(chbuffp, &usr_par, usr_io_map);
	if (ret != LIB_OK)
	{
		return(ret);
	}

	// Inform communication mode to slave and C2 master
	ret = infm_cmode(chbuffp);
	if (ret != LIB_OK)
	{
		return(ret);
	}

	// Check if completed informing communication mode and slave status
	while ((ret = chk_infm_cmode_cmp(chbuffp)) != LIB_OK)
	{
		if (ret == ERROR_INFM_CMODE)
		{
			if (chbuffp->prot_sel == 0)
			{
				// Retry to inform communication mode
				ret = infm_cmode(chbuffp);
				if (ret != LIB_OK)
				{
					return(ret);
				}
			}
			else
			{
				return(ret);
			}
		}
		else if (ret == ERROR_TX_FRM)
		{
			ret = infm_cmode(chbuffp);
			if (ret != LIB_OK)
			{
				return(ret);
			}
		}
	}

	// Start cyclic communication
	if (chbuffp->prot_sel == COM_MODE_SYNC)
	{
		ret = start_sync(chbuffp);
		if (ret != LIB_OK)
		{
			return(ret);
		}
	}
	else
	{
		ret = start_async(chbuffp);
		if (ret != LIB_OK)
		{
			return(ret);
		}
	}

	return ret;
}

/****************************************************************************************/
/* exchange_sync()                                                                      */
```

```
/*                                                                              */
/* Send/receive data processing for sync communication mode.  This will copy   */
/* received data from the communication processor's response buffer to rbuff and */
/* copy data to send from sbuff to the communication processor's command buffer.  */
/********************************************************************************/
short exchange_sync(void)
{
	CHANNEL_INFO*  chbuffp;               // Channel Buffer
	ULONG          rrcvstat[2];           // Receive status
	ULONG          ests;                  // Error Status
	USHORT         st_no;                 // Counter
	short          ret;                   // Return code

	// Initialize value
	ret = OK;

	// Get the pointer of Channel Buffer
	chbuffp = &chbuff;

	// Start cyclic communication
	begin_cyclic_com(chbuffp);

	// Check communication processor status
	ests = read_comstat(chbuffp);

	// Get receive status
	get_rrcvstat(chbuffp, rrcvstat);

	// Iterate through all stations (slaves) and see if they have sent us data.
	for (st_no = 1; st_no <= chbuffp->ma_max; st_no++)
	{
		// Check if we have received any data from this station (slave); if
		// not then continue on with checking next station.  We may also
		// check the error status register at this point.

		// If we detect that we have received data, lets get it.  Copy from
		// the communication processor to our buffer.
		ret = read_ldata(chbuffp, st_no, rbuff[st_no]);
		if (ret != LIB_OK)
		{
			return (ret);
		}

		process_station_data(st_no, rbuff[st_no]);
	}

	// Set up the data for the outgoing Interpolation Feed command.
	for (st_no = 1; st_no <= chbuffp->ma_max; st_no++)
	{
		// Every entry is 4 bytes
		sbuff[st_no][0] = IFEED;              // Constant, 0x34
		sbuff[st_no][1] = target_pos[st_no];  // Update its target position
		sbuff[st_no][2] = vel[st_no];         // Update its vff

		// Since we are doing interpolation feed, we shall be interested in
		// monitoring feedback position.  The monitor code for this is 3.  This
		// accounts for one byte.  Then we have two unused bytes, followed by the
		// watchdog byte.  Regarding the watchdog byte, we compose it with the
		// low 4 bits of our counter combined with the high 4 bits of what this
		// particular station sent us for a watchdog value.
		our_watchdog++;
		sbuff[st_no][3] = (((our_watchdog & 0x0f) + (rcvd_watchdog[st_no] & 0xf0))
<< 24) + 3;
	}

	// Iterate through all stations and if the station had sent data to us
	// then we will send data back to it.
	for (st_no = 1; st_no <= chbuffp->ma_max; st_no++)
	{
```

```
		// If we received something from a station then we send new data to it,
		// otherwise we continue on with checking the next station.

		// If we detect that we need to send, lets send.  Copy from our buffer
		// to the communication processor.
		ret = write_ldata(chbuffp, st_no, sbuff[st_no]);
		if (ret != LIB_OK)
		{
			return (ret);
		}
	}

	return (ret);
}

/*****************************************************************************************/
/* exchange_async()                                                                      */
/*                                                                                       */
/* Send/recieve data processing (async communication mode).                              */
/*****************************************************************************************/
short exchange_async(void)
{
	volatile HOST_IF_REGS *hirp;		// Host I/F Top address
	CHANNEL_INFO*		chbuffp;	// Channel Buffer
	ULONG			ests;		// Error Status
	USHORT			st_no;
	ULONG			ret;		// return code

	// Initialize values
	st_no = 1;
	ret = OK;

	// Set pointer of Channel Buffer
	chbuffp = &chbuff;
	hirp = chbuffp->hif_reg_top;

	// Send async frame
	if (async_sw == ASYNC_SND_SEQ)
	{
		ret = send_frame(chbuffp,
				 DEF_ASYNC_PEER_ADR,
				 DEF_ASYNC_FTYPE,
				 async_sbuff,
				 DEF_ASYNC_DATA_SIZE);

		if (ret == LIB_OK)
		{
			async_sw = ASYNC_RCV_SEQ;
		}
		else if (ret != SNDING_FRAME)
		{
			return(ret);
		}
	}

	// Receive async frame
	if (async_sw == ASYNC_RCV_SEQ)
	{
		ret = req_rcv_frame(chbuffp,
				    &async_rcv_stadr,
				    &async_rcv_stat,
				    &async_rcv_ftype,
				    async_rbuff,
				    &async_rcv_size,
				    (USHORT)DEF_ASYNC_RCV_TOUT_TIME);

		if (ret == RECEIVED_FRAME)
		{
			if ((async_rcv_stat == ASYNC_RCV_CMP) ||
```

```
                    (async_rcv_stat == ASYNC_RCV_TIMOUT))
                {
                        process_async();
                        read_comstat(chbuffp);
                        async_sw = ASYNC_SND_SEQ;
                }
                else
                {
                        // Check communication processor status
                        if ((ests = read_comstat(chbuffp)) != 0)
                        {
                                examine_comstat(chbuffp);
                                return(ERROR_ASIC_STATUS);
                        }
                        return(async_rcv_stat);
                }
        }
        else if (ret != RCVING_FRAME)
        {
                return(ret);
        }
    }
    return(ret);
}

/*****************************************************************************************/
/* exchange_msg()                                                                        */
/*                                                                                       */
/* Send/receive data processing (message communication).                                 */
/*****************************************************************************************/
short exchange_msg(void)
{
    volatile HOST_IF_REGS *hirp;        // Host I/F Top address
    CHANNEL_INFO          *chbuffp;     // Channel Buffer
    USHORT                offset;
    USHORT                ret;          // return code

    // Initialize value
    ret = OK;

    // Set pointer of Channel Buffer
    chbuffp = &chbuff;
    hirp = chbuffp->hif_reg_top;

    // Set send message data size & offset
    offset = 0;

    // Message send sequence
    if (msg_sw == MSG_SND_SEQ)
    {
        // Set send data
        ret = write_msgdata(chbuffp,
                            DEF_C1_MST,
                            offset,
                            DEF_C1MSG_SIZE,
                            c1msg_sbuff);

        if (ret != LIB_OK)
        {
                return(ret);
        }

        // Request send message data
        while ((c1msg_snd_stat = req_snd_msgdata(chbuffp, DEF_C1_MST,
C1MSG_PEER_ADR, DEF_C1MSG_SIZE)) == SNDING_MSG);

        if ((c1msg_snd_stat == ERROR_MSG_ABORT) ||
            (c1msg_snd_stat == ERROR_BUSY_MSG))
        {
```

```
			msg_sw = MSG_SND_SEQ;
		}
		else
		{
			msg_sw = MSG_RCV_SEQ;
		}

		return(c1msg_snd_stat);
	}

	// Message receive sequence
	else if (msg_sw == MSG_RCV_SEQ)
	{
		msg_sw = MSG_SND_SEQ;
		// Request receive message data
		while ((c1msg_rcv_stat = req_rcv_msgdata(chbuffp, DEF_C1_MST,
C1MSG_PEER_ADR, &c1rcv_msgsz)) == RCVING_MSG);

		if (c1msg_rcv_stat == ERROR_MSG_ABORT)
		{
			msg_sw = MSG_RCV_SEQ;
		}

		// Get received messaged data
		if (c1msg_rcv_stat == LIB_OK)
		{
			ret = read_msgdata(chbuffp, DEF_C1_MST, offset, c1rcv_msgsz,
c1msg_rbuff);
			if(ret != LIB_OK)
			{
				return(ret);
			}
		}
	}
	return(ret);
}


/*****************************************************************************************/
/* set_usr_prm()                                                                         */
/*                                                                                       */
/* Set user parameter.                                                                   */
/*****************************************************************************************/
void set_usr_prm(USER_PAR     *usr_parp, USER_IOMAP* usr_iomapp)
{
	USHORT ch;

	// set default user parameter setting
	usr_parp->mod = MOD_TYPE_C1MST | MOD_INT_FR;
	usr_parp->ma0 = 0x0001;                        // My Address(C1 Master:0x0001)
	usr_parp->ma_max = DEF_MA_MAX;
	usr_parp->t_mcyc = DEF_TMCYC;                  // Transmission cycle[10nsec]
	usr_parp->prot_sel = DEF_PROT_SEL;             // sync mode
	usr_parp->max_rtry = DEF_MAX_RTRY;             // Max. number of Retries per
	                                               // Transmission cycle
	usr_parp->wdt = DEF_WDT;                       // Watch dog timer [8usec];
	                                               // if wdt=0, WDT function disabled
	usr_parp->c2_dly = DEF_C2_DLY;                 // C2 delay time
	usr_parp->pkt_sz = DEF_PKT_SZ;
	usr_parp->dly_cnt = 1;                         // System parameter
	usr_parp->intoffset = DEF_INT_OFFSET;          // Interrupt offset time [10nsec]

	// Set IOMAP parameters(C1 master)
	usr_iomapp->axis_adr = 0x0001;                 // Station address
	usr_iomapp->t_rsp = 1000;                      // Transmission response monitoring
	                                               // time [10nsec]
	usr_iomapp->cd_len = 8;                        // Commando data length
	usr_iomapp->rd_len = 8;                        // Response data length
```

```
        // Set IOMAP parameters(slave)
        for (ch = 1; ch <= usr_parp->ma_max; ch++)
        {
                (usr_iomapp+ch)->axis_adr = 0x20 + ch;       // Station address
                (usr_iomapp+ch)->t_rsp = DEF_TRSP;           // Transmission response
                                                             // monitoring time [10nsec]
                (usr_iomapp+ch)->cd_len = DEF_CD_LEN;        // Command data length
                (usr_iomapp+ch)->rd_len = DEF_RD_LEN;        // Response data length
        }
}

/*****************************************************************************************/
/* check_ram()                                                                           */
/*                                                                                       */
/* SRAM read/write check.                                                                */
/*****************************************************************************************/
short check_ram(ULONG *hif_reg_top, USHORT size, ULONG chkdata)
{
        ULONG   work;
        USHORT  ofst, ret;

        // Initialize value
        ofst = 0;
        ret = OK;

        ret = write_ram((ULONG *)HOST_IF_REGS_PTR, 0, size, chkdata);
        if (ret != LIB_OK)
        {
                return(ret);
        }

        while (ofst < DEF_SRAM_SIZE)
        {
                ret = read_ram((ULONG *)HOST_IF_REGS_PTR, ofst, 4, &work);
                if (ret != LIB_OK)
                {
                        return(ret);
                }
                if (work != chkdata)
                {
                        return(ERROR_SRAM_CHECK);
                }
                ofst = ofst + 4;
        }
        return(ret);
}
```