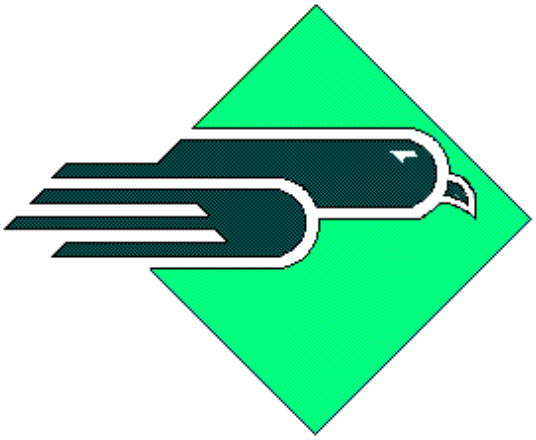

WESTERN AVIONICS

MIL-STD-1553 & STANAG 3910
SOFTWARE DRIVER LIBRARY

P/N 1L01616H01 Rev 5.0

User Manual
UM 01616 Rev M

13/14 Shannon Free Zone
Co. Clare
Ireland

11th July 2002

1. GENERAL INFORMATION

1.1. INTRODUCTION

The software driver package is a library of functions designed to permit easy, flexible software control of one or more of the following Western Avionics MIL-STD 1553 and STANAG 3910 interface boards:

Dual channel 1553 and Stanag 3910 VME board

Dual channel 1553 VME board

1553 and PCI board

Dual 1553 PCI board

1553 and Stanag 3910 PCI board

1553 PC-AT board

1553 and Stanag 3910 PC-AT board

Quad channel 1553 VXI board

Quad channel 1553 and Stanag 3910 VXI board

1553 M-Module in VXI carrier

1553 Bustec module in VXI carrier

These drivers allow configuration of the cards operating mode, with a flexible approach to creating, modifying and managing the fundamental entities which are typically encountered by applications software controlling STANAG 3910 and STANAG 3838 (MIL-STD-1553) data busses. This library is supplied in source code, under Part Number 1L01616H01, to be compiled and linked with the application specific software.

2. INSTALLATION AND PREPARATION FOR USE

2.1. GENERAL

The library is supplied on the disk in IBM PC format. The library is supplied as source code in "C". It is a collection of files, all of which must be copied to a suitable working directory. The user should refer to the compiler vendors manual for a description of the users specific operating system. Refer also to the user manual for the particular card being used.

The following files are contained on disk P/N 1L01616H01:-

cmdrvs.c	setup.c	storefns.h
utils.c	rtfns.c	mrt.h
adrs.c	drivers.c	mrtsamd.h
chkinp.c	msg.c	msg.h
frame.c	mrt.c	rtinit.h
bcmrt.c	rdwr.c	setup.h
hwfns.c	hwfns.h	cmutil.h
rtinit.c	adrs.h	frame.h
cycle.c	chkinp.h	cmdrvs.h
execute.c	allocfns.h	system.h
allocfns.c	cycle.h	rtfns.h
storefns.c	modify.h	drivers.h
mrtsamd.c	errors.h	cflags.h
modify.c	windrvr.h	drivers.def
visa.h	visatype.h	cvi.lib
visa.lib		

2.2. COMPLIER VARIATIONS

The library was designed to be used under ANSI C compilers.

2.3. RESOURCES

The driver library is designed to use the normal run-time library functions available on all systems.

2.4. DOCUMENT CONVENTIONS

The manual documents the functions and data types using the "C" language syntax.
All functions are described using ANSI C prototypes.

3. OPERATION

3.1. INTRODUCTION

The library is used by including the line

#include "drivers.h"

into all application files which need to access any driver function. The linker must link in the following files:-

setup.obj	chkinp.obj
msg.obj	adrs.obj
cycle.obj	modify.obj
frame.obj	rdwr.obj
bcmrt.obj	allocfns.obj
execute.obj	storefns.obj
rtinit.obj	hwfns.obj
mrt.obj	cmdrvs.obj
mrtsamd.obj	cmutil.obj
rtfns.obj	utils.obj

These files will need to have been compiled from their corresponding .c files.
For VXI type cards the library files CVI.LIB and VISA.LIB will also be required.

3.2. FUNCTIONS

The library functions are divided into four categories. The full list of functions is as follows:-

Initialisation functions

setup( )	Initialise the card
rdSetup( )	Examine the initialisation information
initPC()	Initialise the card for PC operation

Bus Controller functions

modBcMrtRt( )	Modify an RT
rdBcMrtRt( )	Examine an RT
crMsg( )	Create a Message entity
modMsg( )	Modify a Message entity
rdMsg( )	Examine a Message entity
joinMsg( )	Link two messages to same buffer
linkMsg( )	Form message with link buffers
insertMsg( )	Insert a BC message
crCycle( )	Create a Cycle entity
modCycle( )	Modify a Cycle entity
rdCycle( )	Examine a Cycle entity
crFrame( )	Create a Frame entity
modFrame( )	Modify a Frame entity
rdFrame( )	Examine a Frame entity
runFrame( )	Start the BC transmission
haltFrame( )	Stop the BC transmission

Multi-Remote Terminal functions

modMrtRt( )	Modify an RT
rdMrtRt( )	Examine an RT
crMrtRtSa( )	Create a Sub-Address
modMrtRtSa( )	Modify a Sub-Address
rdMrtRtSa( )	Examine a Sub-Address
modMrtRtMd( )	Modify a LS Mode Code
rdMrtRtMd( )	Examine a LS Mode Code
srqQueue( )	Generate a service request in queue
joinMrt( )	Join two RT sub addresses to same buffer
linkMrt( )	Form link list buffers for a sub address
extendMrt( )	Form extended sub address for RT
actMrt( )	Activate the RTs
deActMrt( )	Deactivate the RT

Chronological Monitor functions

defTrigs( )	Define the triggers
defSeq( )	Define the trigger sequence
getRange( )	Find range of captured messages
getMsg( )	Read a message from the stack
findMsgs( )	Search the stack for a message
fmtMsgL( )	Covert message into text string (long)
fmtMsgM( )	Covert message into text string (medium)
fmtMsgS( )	Covert message into text string (short)

General Functions

rdStatusReg( )	Examine the card's Status Register
setClk( )	Set local clock value
readClk( )	Read local clock value
rdBCInfo( )	Read BC pointer info
rdRTinfo( )	Read RT pointer
rdQueue( )	Read queue information
fileInfo( )	Create text file defining all pointers
exCmd( )	Execute a command
isCardPresent( )	Checks if card is present in system
wWord( )	Write a 16 bit word to the card
rWord( )	Read a 16 bit word from the card
selfTest( )	Execute self-test
drvsDebug()	Drivers visual debugger for console mode

Many other functions are used to implement these drivers. All of these function names start with the characters _w. When developing application code the user must create function names whose first two characters are not _w.

3.3. DATA TYPES

The file drivers.h defines the data types used by the library driver functions. All data types begin with an uppercase letter and all functions with a lowercase letter. To overcome the differences in the size of the pre-defined "C" data types (e.g., on some compilers the "int" is 16-bits, on others 32-bits) the file system.h included automatically by drivers.h provides typedefs for bytes and words. This file may need to be changed for certain compilers.

The following data types are used consistently throughout the driver code:

Ubyte	unsigned 8-bit value
Sbyte	signed 8-bit value
Uword	unsigned 16-bit value
Sword	signed 16-bit value
Ulong	unsigned 32-bit value
Slong	signed 32-bit value
Bool	TRUE/FALSE Boolean value, defined as Sword
Error	unsigned 16-bit value
NO_PTR	(Ulong)NULL

Throughout the drivers, the parameters which are strictly "enumerations" where the value is used as a constant to identify non-numeric value. For example, the setup( ) function takes the card type which identifies whether a VXI, VME, PCI card etc. is being used) as a Uword. No mathematical function is ever performed on these, only a comparison for equality or inequality.

All functions return a type '**Error**'. If the function is executed successfully the value of this Error will be **E_NO_ERROR**. Otherwise the value will be one of the values listed in the appendix.

3.4. PLATFORM SELECTION

The drivers will compile for the platform as defined in cflags.h. These platforms are:

PLATFORM_PCI_PCAT	Compile drivers for PCI/PC-AT card
PLATFORM_VXI	Compile drivers for VXI card
PLATFORM_VME	Compile drivers for VME environment
PLATFORM_BUG	Compile drivers for DEBUG (no card access)

Note:

- Each platform has individual flags that can be set in cflags.h. If the user wishes to compile the drivers to create a DLL the flag CREATE_DLL_FILE must be true.

- Care must be taken to ensure the correct card type is chosen. For further details see initPC() and setup().

3.5. DYNAMIC LINK LIBRARIES

To compile the drivers into a dynamic link library the file drivers.c must be used as the main file. This is simply a file with all the drivers in the library converted to the format required for compiling a DLL. Each function in drivers.c is appended with the letters DLL and given a prototype that is compatible with a DLL function. This function will call the actual driver function as defined this library. For example:

actMrtDLL( ) in drivers.c → calls actMrt( )

Any application calling the DLL must call actMrtDLL( ) and not actMrt( ). The file drivers.def is also included as the definition file for compiling the drivers for a DLL. It is recommended that, for PCI and VXI platforms, the user create a DLL and use this library to call particular functions for the application.

The files VXI_DRVS.DLL, VXI_DRVS.LIB, PC_DRVS.DLL and PC_DRVS.LIB are provided to allow immediate use of the drivers in any windows application.

4. INITIALISATION FUNCTIONS

4.1. INTRODUCTION

There are three initialisation functions, initPC( ) setup( ) and rdSetup( ). The function initPC( ) is only used when the drivers are to be used in a PCI or VXI environment. The function setup( ) must be used for the VME environment. Failure to do this will result in unpredictable results for any later function calls.

4.2. initPC( )

Error initPC (MemMapping *cardHandle, Pcinfo *info)

Parameter	**Description**
cardHandle	This is a pointer to an un-initialised MemMapping structure
info	This is a pointer to a PCinfo structure defining the various PC related setup requirements

The elements of the Pcinfo structure are as follows:

Element	**Description**
cardType	This **must** be set to the correct card type the user wishes to target: VXI_CARD (Quad channel 1553 and/or Stanag 3910 VXI card) VXIM_CARD (M-Module 1553 card) VXIB_CARD (Bustec 1553 module) PCI_CARD (PCI 1553 and/or Stanag 3910 card) PCAT_CARD (PC-AT 1553 and/or Stanag 3910 card) VME_CARD (Dual channel 1553 and/or Stanag 3910 VME card)
lsCoupling	This shall define the 1553B coupling and shall be set to one of 2 values: DIRECT_COUPLING STUB_COUPLING
amplitude	This shall be set to a value 0x00-0xFF defining the DAC value to be set for the TX amplitude.
busModeVXI	This is only applicable to Western Avionics VXI quad channel cards and shall be set to one of 2 values: BUS_MODE_2 – connect 1553 of modules 0,1 and 2,3 BUS_MODE_4 – leave all four 1553 modules separate
VXIaddr	This shall be set to a value 0x00-0xFF defining the logical address of the VXI card.
addSpace	This shall be set to address space setting for the VXI card. This shall be set to one of 2 values: ASPACE_24 ASPACE_32

VXImoduleNo

This is only applicable to Western Avionics VXI quad channel cards and BUSTEC modules. This shall be set to the module number for configuration and selection.

For Western Avionics quad channel cards:
If the MSB of this 16 bit parameter is set, then full initialisation of the module will be carried out. If the card has already been initialised and the user simply needs to select the module, the MSB must be left clear.

Example: 0x8000 – Initialise and select module 0
0x0002 – Select module 2 without initialisation

For BUSTEC modules:
The value of this must be in the range 0-7. The MSB must never be set.

Example: 0x0000 – Select the first of 8 modules in the BUSTEC carrier.

ClockType

This shall be set to one of two values:

CLOCK_IRIGB : Module uses IRIG-B type clock
CLOCK_32BIT : Module uses standard 32 bit clock

pciCardNo

This shall define the particular PCI card to be initialised within the PCI rack. It is possible to have a number of PCI cards within the rack. The drivers will scan the rack and create a list of available cards in ascending slot order. The value of pciCardNo will define which card is to be initialised.
This value is a logical card number from 1 to n where 'n' is the number of cards in the rack. The value 0 is an invalid value.
Eg: Two cards at slots 3 and 5 will be pciCardNo values 1 and 2 respectfully.

pciModuleNo

This shall select the module number to be addressed on the PCI card. This is only relevant for PCI cards that have more than one 1553/3910 module. A value of 0 will address the 1st module. For single module cards this value **must** be set to 0.

Note:

- For Western Avionics VXI quad channel type cards all parameters are required. If the function is used for initialisation (VXImoduleNo MSB = 1), a VI instrument handle will be created for the card and saved in the structure MemMapping (MemMapping->instrHndl).
- For PCI cards, pciCardNo, amplitude and lsCoupling are the only parameters required.

Description:

The initPC( ) function prepares the MemMapping function for use in a PC environment.

Example:

```
#include "drivers.h"
…
static    MemMapping card1;
Error     error;
Pcinfo    info;
…
info.pciCardNo          = 1;
info.pciModuleNo        = 0;
info.lsCoupling         = DIRECT_COUPLING;
info.amplitude          = 0x80;
error = initPC (&card1, &info);
```

4.3. setup( )

Error setup (MemMapping *cardHandle, Uword *info,
Sword cardType, Sword opMode, Ulong cardAddress)

Parameter	**Description**
cardHandle	This is a pointer to an un-initialised MemMapping structure
Info	This is a pointer to an array of SZ_SetupInfo unsigned 16-bit quantities, each of which allows a default parameter to be set explicitly.
cardType	This **must** be set to the correct card type the user wishes to target: VXI_CARD, VXIM_CARD, VXIB_CARD, PCI_CARD, PCAT_CARD, VME_CARD
opMode	This defines the operating mode of the target card. Permitted values are:- BCMRT_MODE Bus Controller and simultaneous multi-RT MRT_MODE Multiple RT CM_MODE Chronological Monitor Mode
cardAddress	This parameter is for VME platforms only. This shall be set to the absolute VME address the card resides. The card access functions rWord() and wWord() cast this value as a word pointer to the start of the card. In a lot of cases the operating system will not allow this type of access. If this is the case then it is up to the user to modify the initialisation and card access functions to incorporate any system specific drivers for accessing the card.

Description:

The setup ( ) function prepares the physical board for use by the driver library. It sets default values, some of which may be modified by the info[ ] data (shown below). The data structure of type MemMapping is filled with essential information by setup ( ) and must not be modified by the application code. The pointer to this structure is used as a "handle" by the other driver functions to uniquely identify the board.

The info [ ] array contains user defined defaults. The first element (info [0]) is a mask, organised on a bit basis defining which elements of the info[ ] array are present. A NULL pointer can be used to indicate that all the defaults should be used.

The fields of info [0], which may be OR'd are:-

F_HsSubAddress	High Speed Sub-address. Info[HsSubAddress] contains the new HS sub-address. When not set the value is 26 decimal.
F_VMEIrq1	VME Irq1 definition. info[VMEIrq1] defines VME interrupt 1. When not set the value is 0 (Irq 1 undefined). This is used by BCMRT, MRT and CM modes.
F_VMEIrq2	VME Irq2 definition. info[VMEIrq2] defines VME interrupt 2. When not set the value is 0 (Irq 2 undefined). This is used by BCMRT, MRT and CM modes.
F_IRQSeln	IRQ selection. info[IRQSeln] allows the user to select the interrupt. When not set the value is 0 (no interrupts selected). This is used by BCMRT, MRT and CM modes.
F_RtResponseTm	RT Response time. Info[RtResponseTm] contains the new RT response time in uS. When not set the value is 6 uS. This is not used if opMode is CM_MODE.
F_BcCycleTm	The cycle length. Info[BcCycleTm] contains the new cycle time in multiples of 10 uS. When not set the value is 2000 decimal (equivalent to a cycle time of 20mS). This is only used when opMode is in BCMRT_MODE.

cardType is a signed 16-bit quantity. If omitted it will default to PCI_CARD

opMode is a signed 16-bit quantity. This parameter is mandatory.

cardAddress is a 32-bit unsigned quantity. This parameter is mandatory for VME platforms.

Example:

```
#include "drivers.h"
…
static MemMapping card1;
Error  error;
…

error = setup (&card1, NULL, VME_CARD, BCMRT_MODE, 0xC00000)
```

This sets up a VME card at VME address 00C00000 Hex in BC/MRT mode with the default parameters. The pointer &card1 can then be used to identify this card to other driver functions. The setup ( ) function can be called again to change the operating mode if desired. Note that each call to setup ( ) reinitialises the card.

4.4. rdSetup( )

Error rdSetup (MemMapping *cardHandle, Uword *info,
Sword *cardType, Sword *opMode, Ulong *cardAddress)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initialised by a call to setup( ).
info	Pointer to an array of size SZ_SetupInfo unsigned 16-bit quantities which will be filled in by rdSetup( ) with the setup information. See below.
cardType	This is a pointer to a signed 16-bit quantity which is filled in by rdSetup( ). It will be set to: VXI_CARD, VXIM_CARD, VXIB_CARD, PCI_CARD, PCAT_CARD or VME_CARD.
opMode	This is a pointer to a signed 16-bit quantity which is filled in by rdSetup( ). It will be set to :-
cardAddress	This is a pointer to an unsigned 32-bit quantity which is filled in by rdSetup( ). It is the absolute physical VME address at which the card is located.

BCMRT_MODE	Bus Controller & Simultaneous multi-RT
MRT_MODE	Multiple RT
CM_MODE	Chronological Monitor Mode

Description:

The rdSetup( ) function reads the setup information from the card identified by cardHandle. The info[ ] array holds the returned parameters. Should info be NULL then these fields will not be filled in.

The elements of info[ ] which contain valid data are:-

HsSubAddress	High Speed Sub Address. info[HsSubAddress] contains the HS Sub Address.
VMEIrq1	VME Irq 1. info[VMEIrq1] contains the definition for VME Interrupt 1.
VMEIrq2	VME Irq 2. info[VMEIrq2] contains the definition for VME Interrupt 2.
IRQSeln	IRQ Selection. info[IRQSeln] contains the interrupt selection word.

RtResponseTm	RT Response Time. info[RtResponseTm] contains the RT Response Time in uS. This will be indeterminate if the operating mode is CM_MODE.
BcCycleTm	The Cycle length. info[BcCycleTm] contains the cycle time in multiples of 10 uS. This is only relevant when the operating mode is BCMRT_MODE.
BcTmOut	The Bus Controller timeout. info[BcTmOut] contains the BC timeout in uS. This is only relevant when the operating mode is BCMRT_MODE.

cardType is a pointer to a signed 16-bit quantity. Should cardType be NULL then this parameter is not filled in by rdSetup( ).

opType is a pointer to a signed 16-bit quantity. Should opType be NULL then this parameter is not filled in by rdSetup( ).

cardAddress is a pointer to a unsigned 32-bit quantity. Should cardAddress be NULL then this parameter is not filled in by rdSetup( ).

Example:

```
#include "drivers.h"
...
Uword   info1[SZ_SetupInfo];
Sword   opMode;
Sword   cardType;
Ulong   cardAddress;
Error   error;

error = rdSetup (&card1,info1,&cardType,&opType,&cardAddress);
```

5. BUS CONTROLLER FUNCTIONS

5.1. INTRODUCTION

The Bus Controller functions manage the setup of the card and control the transmission of data when in BC/MRT mode. The drivers organise the Bus controller as a "Frame" which contains one or more "Cycles". Each cycle contains one or more "Messages". The messages define the data transfer in terms of the source, destination, number of words, bus(es) used, gap times, error injection and the data values to be transmitted. A "Cycle" is a sequence of messages and is transmitted in a fixed time, known as the cycle time. The "Frame" is a list of "Cycles" and is transmitted one or more times by the runFrame( ) function. The modBcMrtRt( ) driver must be called to setup all the simulated RT's before the crMsg( ) driver is called.

5.2. modBcMrtRt( )

Error modBcMrtRt (MemMapping *cardHandle, Sword rtNum, Uword *info)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initialised by a call to setup( ).
rtNum	The RT to be modified.
info	Pointer to an array of size SZ_BcMrtRtInfo unsigned 16-bit quantities which modify an RT.

Description:

The modBcMrtRt( ) function modifies the configuration of an RT in the card identified by cardHandle. It is essential that this has been initialised via a previous call to setup( ). The MRTs automatically default as follows during setup( ):

Element	Default Value
RtState	DISABLED_RT
RtStatus	RT 0 0x0000; RT 1 0800 hex; RT 2 1000 hex etc.
RtVector	0
RtLsBitWord	0
RtHsBitWord	0

rtNum is a signed 16-bit quantity. It is the RT, which is to be modified, and the user must set it. It must contain a valid RT number. This parameter is MANDATORY. info[ ] defines the modification data for the RT. The first element is a mask organised on a bit basis defining which elements of the info[ ] array are present. The fields of info[0], which may be OR'd together are:-

F_RtState	RT State. info[RtState] contains the simulation state of the card. It must be one of the following:	
	SIMULATED_RT	The RT is simulated by the card (An INTERNAL RT)
	DISABLED_RT	The RT is off-card (An EXTERNAL RT)
F_RtStatus	RT Status. info[RtStatus] contains the status word, which will be transmitted by the RT when the RT is simulated.	
F_RtVector	RT Vector Word. info[RtVector] contains the vector word, which will be transmitted by the RT when the Transmit Vector Word mode code command is received (and the RT is simulated).	
F_RtLsBitWord	RT LS BIT Word. info[RtLsBitWord] contains the LS BIT word, which will be transmitted by the RT when the Transmit Bit Word mode code command is received (and the RT is simulated).	
F_RtHsBitWord	RT HS BIT Word. info[RtHsBitWord] contains the HS BIT word which will be transmitted by the RT when a transmit command is sent to the HS Sub Address (and the RT is simulated).	

Example:

```
#include "drivers.h"

...

Uword   info1 [SZ_BcMrtRtInfo] ;
Sword   rtNum;

rtNum = 1 ;

/*
Simulate RT 1 and use the default settings
*/
info1[0]                = F_RtState ;
info1[RtState]          = SIMULATED_RT;

error = modBcMrtRt (&card1,rtNum,info1) ;
...
/*
  With the card running modify the vector word and the LS BIT word
*/
info1[0]                = F_RtVector | F_RtLsBitWord ;
info1[RtVector]         = 0x1234 ;
info1[RtLsBitWord] = 0xABCD ;

error = modBcMrtRt (&card1,rtNum,info1) ;
```

5.3. rdBcMrtRt( )

Error rdBcMrtRt (MemMapping *cardHandle, Sword rtNum, Uword *info)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initialised by a call to setup( ).
rtNum	The RT to be examined.
info	Pointer to an array of size SZ_BcMrtRtInfo unsigned 16-bit quantities which will be filled in by rdBcMrtRt( ) with the RT information. See below.

Description:

The rdBcMrtRt( ) function reads the configuration information of an RT in the card identified by cardHandle. It is essential that this has been initialised via a previous call to setup( ). rtNum is a signed 16-bit quantity. It is the RT, which is to be read, and the user must set it. It must contain a valid RT number. This parameter is MANDATORY. info[ ] holds the data read from the RT. Should info be NULL then these fields will not be filled in. The elements of info[ ] are:-

RtState — RT State. info[RtState] contains the simulation state of the card, which is one of the following:-

SIMULATED_RT	RT is simulated by the card (An INTERNAL RT)
DISABLED_RT	RT is off-card (An EXTERNAL RT)

RtStatus — RT Status. info[RtStatus] contains the status word, which will be transmitted by the RT (when the RT is simulated).

RtVector — RT Vector Word. info[RtVector] contains the vector word, which will be transmitted by the RT when the Transmit Vector Word mode code command is received (and the RT is simulated).

RtLsBitWord — RT LS BIT Word. info[RtLsBitWord] contains the LS BIT word, which will be transmitted by the RT when the Transmit Bit Word mode code command is received (and the RT is simulated).

RtHsBitWord — RT HS BIT Word. info[RtHsBitWord] contains the HS BIT word which will be transmitted by the RT when a transmit command is sent to the HS Sub Address (and the RT is simulated).

Example:

```
#include "drivers.h"

...

Uword   info1[SZ_BcMrtRtInfo] ;
Sword   rtNum;
Error = error

rtNum = 1 ;

/*
   Examine the configuration of RT 1
*/
error = rdBcMrtRt (&card1,rtNum,info1) ;
```

5.4. crMsg( )

Error crMsg (MemMapping *cardHandle, Slong *msgId,
Uword *info, DataInfo, msgData)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
msgId	pointer to a variable which crMsg fills in with the identifier of the message created.
info	Pointer to an array of SZ_MsgInfo unsigned 16-bit quantities which define the message to be created. See below.
msgData	Pointer to a DataInfo structure (shown below) which defines the data to be associated with the message.

Description

The crMsg( ) function creates a new message in the card identified by cardHandle. It is essential that this has been initialised via a previous call to setup( ). crMsg assigns a unique identifier to the message, returning this value in the variable pointed to by msgId. The info[ ] array defines the message. The element (info [0]) is a mask, organised on a bit basis defining which elements of the info[ ] array are present.

The fields of info[ ], which may be OR'd together are:-

F_MsgType Message Type. Info[MsgType] contains the type of message. This field is mandatory and must be one of the following:

BC_RT_3838	BC to RT low speed only.
RT_BC-3838	RT to BC low speed only.
RT_RT_3838	RT to RT low speed only.
B_BC_RT_3838	Bcast BC to RT low speed only.
B_RT_BC_3838	Bcast RT to BC low speed only.
B_BC_RT_CLK_3838	Bcast RX clock message.
TXMD_WODA_3838	3838 TX Mode no data.
TXMD_WIDA_3838	3838 TX Mode with data
RXMD_WIDA_3838	3838 RX Mode with data
B_MODE_WODA_3838	Bcast 3838 Mode no data
B_RXMD_WIDA_3838	Bcast 3838 Mode with data
BC_RT_3910	BC to RT with HS data.
RT_BC_3910	RT to BC with HS data.
RT_RT_3910	RT to RT with HS data.
B_BC_RT_3910	Bcast BC to RT with HS data.
B_RT_RT_3910	Bcast RT to RT with HS data.

	TX_MSG_3910	RT to BC low speed from the RT's high speed sub-address (HS status/1st action word/bit words)
	MODE_3910	HS Mode Command
	B_MODE_3910	Bcast HS Mode Command.

Note:
If the cardType chosen in setup( ) is not a 3910 type then all message types which are _3910 are irrelevant

F_LsIMsgGap Inter Message Gap. info[LsIMsgGap] contains the gap time, in multiples of 0.1 uS, after the message before the following message will be transmitted. If omitted, crMsg( ) uses 100 uS as the default value.

F-LsBus Low Speed Bus. info[LsBus] contains the low speed bus to be used. This is either PRIMARY_BUS or SECONDARY_BUS. If omitted, crMsg( ) uses PRIMARY_BUS as the default value.

F_RT1 RT Number. info[RT1] contains the number of the RT involved in the message. For RT to RT transfers this is the RT that receives the data. If omitted, crMsg( ) uses 1 as the default value.

F_SubAdrs1 Sub Address. For low speed transfers, info[SubAdrs1] contains the sub-address of RT1 used in the transfer. For high speed transfers, info[SubAdrs1] contains the high speed identifier (sub-address) of the RT used in the transfer. If omitted, crMsg( ) uses 1 as the default value. For low speed Mode Codes the value of info[SubAdrs1] will automatically default to zero. The user can set it to 31 decimal if required.

F_RT2 RT Number. info[RT2] contains the number of the transmitting RT for RT to RT transfers only, and is ignored for all other transfer types. If omitted, crMsg( ) uses 2 as the default value for the transmitting RT.

F_SubAdrs2 Sub Address. For low speed transfers, info[SubAdrs2] contains the sub-address of RT2 used in the transfer. For high speed transfers, info[SubAdrs2] contains the high speed identifier (sub-address) of the RT used in the transfer. If omitted, crMsg( ) uses 2 as the default value.

F_WCnt Word count. For low speed transfers, info[WCnt] contains the number of words to be transferred. For high speed transfers, info[WCnt] contains the number of words in the INFO field of the high speed frame to be transferred. If omitted, crMsg( ) uses 32 as the default value.

F_ModeCode Mode Code. For low speed Mode Commands, info[ModeCode] contains the mode code number of the required low speed mode command. If omitted, crMsg( ) uses 2 as the default value. For high speed Mode Commands, info[ModeCode] contains the mode code number of the required high speed mode command. If omitted, crMsg( ) uses 3 as the default value.

F_LsErrors Low speed Error Injection. info[LsErrors] contains the type of error to be injected in the transfer. This must be one of the following:

NO_LS_ERRS	LS Error Injection is disabled.
PARITY_ERR	Inject a parity error.
MANCHESTER_ERR	Inject a Manchester error
SYNCHRO_ERR	Inject a synchro. error
WRD_LEN_ERR	Inject a word length error
WRONG_BUS_ERR	Inject a wrong bus error
BOTH_BUS_ERR	Transmit on both buses
RESP_TM_ERR	Inject a response time error
POS_WRD_CNT_ERR	Inject a +ve word count error
NEG_WRD_CNT_ERR	Inject a -ve word count error

If omitted, crMsg( ) uses NO_LS_ERRS as the default value.

F_LsErrPhase Low speed Error Phase. info[LsErrPhase] determines which part of the message will have the error. Except in the case of NO_LS_ERRS it must be set to one of the following:

ERR_DISABLED	No error injection
ERR_FIR_BC_TX	Error injected into command or data word transmitted by BC.
ERR_SEC_BC_TX	Error injected into second command or action word of a high speed RT to RT transfer.
ERR_FIR_RT_TX	Error injected into RT response message, status or data, when the card simulates the RT.
ERR_SEC_RT_TX	Error injected into second RT status word of an RT to RT transfer when the transmitting RT is simulated by the card.

F_LsErrInfo Low speed Error Information. info[LsErrInfo] contains additional information, which depends on the value of info[LsErrors]

info[LsErrors]	info[LsErrInfo]
MANCHESTER_ERR	Bit position in the word where error is injected. Zero is the most significant bit.
SYNCHRO_ERR	The desired sync. pattern. This is 6 bits each representing 0.5 uS of the sync. period. A '1' forces the sync. high, '0' forces it low. A pattern of 0 is not allowed.
WRD_LEN_ERR	The desired word length in bits. A value of 16 is the "normal" word length, 15 would be a word short by 1 bit.
RESP_TM_ERR	The required RT response time in microseconds.
POS_WRD_CNT_ERR	The number of extra words transmitted in the message.
NEG_WRD_CNT_ERR	The number of words to be omitted from the transmitted message.

F_LsErrPosn Low speed Error Position. info[LsErrPosn] contains the word number where the error will be injected when the info[LsErrors] is PARITY_ERR, SNCHRO_ERR or WRD_LEN_ERR. Word zero is the command or status word, depending on the setting of info[LsErrPhase].

F-HsRtRtIMsgGap RT to RT Gap Time info[HsRtRtIMsgGap] contains the gaptime in uS between the two low speed parts (Command/Action pairs) of a high speed RT to RT transfer. This should only be used when info[MsgType] specifies a high speed RT to RT transfer. If omitted, crMsg( ) uses 20 decimal as the default value.

F_HsBus High Speed Bus. info[HsBus] contains the high speed bus to be used. This is either PRIMARY_BUS or SECONDARY_BUS. This should only be used when info[MsgType] specifies a high speed transfer. If omitted, crMsg( ) uses PRIMARY_BUS as the default value.

F_HsErrors

High Speed Error Injection. This should only be used when info[MsgType] specifies a high speed transfer. info[HsErrors] contains the type of error to be injected in the transfer. If it is set to NO_HS_ERRS then high speed error injection is disabled, otherwise one or a combination of high speed errors may be injected as follows:

PRE_BIT_CNT_ERR

The number of preamble bits, which is normally set to 40, is set to the value contained in info[HsErrInfo]

NEG_3910_WRD_CNT_ERR

The high speed frame is transmitted with one word less than normal.

POS_3910_WRD_CNT_ERR

The high speed frame is transmitted with one extra word appended.

FCS_ERR

The frame check sequence is transmitted with an invalid value.

HS_BIT_CNT_ERR

The number of bits to be removed from the HS data stream to create a bit count error. The number of bits to be removed is a value of 0-15 as defined in info[HsErrInfo]

HS_SD_ED_ERR

The contents of info[HsErrInfo] defines the bit pattern to be used for the start delimiter and end delimiter. The most significant byte defines the end delimiter and the least significant byte defines the start delimiter. For a good start and end delimiter this value shall be 0x8E71.

GATE_ERR — The single bit of the high speed frame is transmitted without a Manchester transition, such that it is low throughout the bit time. info[HsHErrPosn] and info[HsLErrPosn] specify the high and low order words of the bit position where this occurs. Bit 0 is the first bit of the preamble. If GATE_HIGH is OR'd with info[HsHErrPosn] then the bit will be transmitted high throughout the bit time.

If omitted, crMsg( ) uses NO_HS_ERRS as the default value.

F_HsRiTiTmOut — High Speed Receiver/Transmitter Initialise Time. info [HsRiTiTmOut] is only used when info[MsgType] specifies a high speed transfer. This element should be set up as a Receiver Initialise Time or Transmitter Initialise Time as appropriate for the high speed transfer type. If omitted crMsg( ) will set it up as a receiver or transmitter time as appropriate, defaulting to 200uS for receiver and 24uS for transmitter

msgData is a pointer to a DataInfo structure. Should msgData be NULL, then a default un-initialised data buffer is allocated for the message. The size is sufficient to hold the number of words specified by info[WCnt] as appropriate based on the value of info[MsgType]

The DataInfo type is declared as:

```
typedef struct  { Sword  size;
                  Sword  posn;
                  Sword  action;
                  Uword  *data;
                  }  DataInfo;
```

size — Specifies the size in words of the data buffer to be created. crMsg( ) automatically accounts for the extra header words in a high speed message. A value of zero specifies that the default size should be used as if msgData was NULL.

posn — Is unused.

action — Specifies the action on the data buffer. This must be set to NEW in order to create a new data buffer.

data — Is a pointer to an array of words which are copied into the newly created data buffer. If data is NULL then the data buffer contents are set to zero. At most, 'size' words are copied into the data buffer.

NOTE: Only the data words can be setup by the crMsg( ) driver. The time tag words default automatically. In the case of a high speed message the FCPA, DA and WC words also default automatically.

Example:

```
#include "drivers.h"

…

Slong           msg1, msg2;
Uword           info1[SZ_MsgInfo];
Uword           info2[SZ_MsgInfo];
DataInfo        datainfo;
Uword           data1[32]
Sword           i;
Error           error;

/*
  Set info1 for a BC to RT HS Message and
  info2 to get the HS status of the message
*/
info1[0] = F_MsgType | F_LsIMsgGap | F_LsBus | F_RT1 | F_WCnt |  F_HsBus;
info1[MsgType]          = BC_RT_3910;
info1[RT1]              = 1;
info1[SubAdrs1]         = 1;
info1[LsBus]            = PRIMARY_BUS
info1[HsBus]            = PRIMARY_BUS
info1[WCnt]             = 32;
info1[LsIMsgGap]        =25*10;

info2[0]  = F_MsgType | F_LsBus | F_RT1 | F_WCnt |
info2[MsgType]          = TX_MSG_3910;
info2[RT1]              = 1;
info2[SubAdrs1          = 26;
info2[LsBus]            = SECONDARY_BUS;
info2[WCnt]             = 1;

for  (i=0; i,32 ; i++)
      data1[i]  =  0;

datainfo.size           = 32;
datainfo.action         = NEW;
datainfo.data           = data1;

/*
    Create a message using info1 with initialised data buffer
    Create a message using info2 but leave data buffer un initialised
*/
error = crMsg (&card1, &msg1, info1, &datainfo);
error = crMsg (&card1, &msg2, info2, NULL);
```

5.5. modMsg( )

Error crMsg (MemMapping *cardHandle, Slong msgId,
Uword *info, DataInfo, msgData)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
msgId	id of previously created message
info	Pointer to an array of SZ_MsgInfo unsigned 16-bit quantities which defines the changes to the already created message.
msgData	Pointer to a DataInfo structure which defines the data to be associated with the message.

Description

The modMsg( ) function modifies an existing message in the card identified by cardHandle. modMsg uses the unique identifier assigned by a previous call to crMsg( ) to identify the message to be modified. The info[ ] array defines the parameters of the message to be modified. The first element (info [0]) is a mask, organised on a bit basis defining which elements of the info[ ] array are present. If info is NULL, then only the data specified by msgData is modified, all other parameters of the message remaining unchanged.

The fields of info[ ], which may be OR'd together are:-

F_MsgType Message Type. Info[MsgType] contains the type of message. This field is mandatory and must be one of the following:

BC_RT_3838	BC to RT low speed only.
RT_BC-3838	RT to BC low speed only.
RT_RT_3838	RT to RT low speed only.
B_BC_RT_3838	Bcast BC to RT low speed only.
B_RT_BC_3838	Bcast RT to BC low speed only.
B_BC_RT_CLK_3838	Bcast RX clock message.
TXMD_WODA_3838	3838 TX Mode no data.
TXMD_WIDA_3838	3838 TX Mode with data
RXMD_WIDA_3838	3838 RX Mode with data
B_MODE_WODA_3838	Bcast 3838 Mode no data
B_RXMD_WIDA_3838	Bcast 3838 Mode with data
BC_RT_3910	BC to RT with HS data.
RT_BC_3910	RT to BC with HS data.
RT_RT_3910	RT to RT with HS data.
B_BC_RT_3910	Bcast BC to RT with HS data.
B_RT_RT_3910	Bcast RT to RT with HS data.

TX_MSG_3910	RT to BC low speed from the RT's high speed sub-address (HS status/1st action word/bit words)
MODE_3910	HS Mode Command
B_MODE_3910	Bcast HS Mode Command.

Note:
If the cardType chosen in setup( ) is not a 3910 type then all message types which are _3910 are irrelevant

F_LsIMsgGap Inter Message Gap. info[LsIMsgGap] contains the gap time, in multiples of 0.1 uS, after the message before the following message will be transmitted.

F-LsBus Low speed Bus. info[LsBus] contains the low speed bus to be used. This is either PRIMARY_BUS or SECONDARY_BUS.

F_RT1 RT Number. info[RT1] contains the number of the RT involved in the message. For RT to RT transfers this is the RT that receives the data.

F_SubAdrs1 Sub Address. For low speed transfers, info[SubAdrs1] contains the sub-address of RT1 used in the transfer. For high speed transfers, info[SubAdrs1] contains the high speed identifier (sub-address) of the RT used in the transfer. For low speed Mode Codes the value of info[SubAdrs1] contains either 0 or 31 decimal.

F_RT2 RT Number. info[RT2] contains the number of the transmitting RT for RT to RT transfers only, and is ignored for all other transfer types.

F_SubAdrs2 Sub Address. For low speed transfers, info[SubAdrs2] contains the sub-address of RT2 used in the transfer. For high speed transfers, info[SubAdrs2] contains the high speed identifier (sub-address) of the RT used in the transfer.

F_WCnt Word count. For low speed transfers, info[WCnt] contains the number of words to be transferred. For high speed transfers, info[WCnt] contains the number of words in the INFO field of the high speed frame to be transferred. If omitted, crMsg( ) uses 32 as the default value.

F_ModeCode Mode Code. For low speed Mode Commands, info[ModeCode] contains the mode code number of the required low speed mode command The value of info[SubAdrs1] contains either 0 or 31 decimal for a low speed mode code transfer. For high speed Mode Commands, info[ModeCode] contains the mode code number of the required high speed mode command.

F_LsErrors Low speed Error Injection. info[LsErrors] contains the type of error to be injected in the transfer. This must be one of the following:

NO_LS_ERRS	LS Error Injection is disabled.
PARITY_ERR	Inject a parity error.
MANCHESTER_ERR	Inject a Manchester error
SYNCHRO_ERR	Inject a synchro. error
WRD_LEN_ERR	Inject a word length error
WRONG_BUS_ERR	Inject a wrong bus error
BOTH_BUS_ERR	Transmit on both buses
RESP_TM_ERR	Inject a response time error
POS_WRD_CNT_ERR	Inject a +ve word count error
NEG_WRD_CNT_ERR	Inject a -ve word count error

info[LsErrPhase] determines which part of the message will have the error. Except in the case of NO_LS_ERRS it must be set to one of the following:

ERR_DISABLED	No error injection
ERR_FIR_BC_TX	Error injected into command or data word transmitted by BC.
ERR_SEC_BC_TX	Error injected into second command or action word of a high speed RT to RT transfer.
ERR_FIR_RT_TX	Error injected into RT response message, status or data, when the card simulates the RT.
ERR_SEC_RT_TX	Error injected into second RT status word of an RT to RT transfer when the transmitting RT is simulated by the card.

info[LsErrInfo] contains additional information which depends on the value of info[LsErrors]

info[LsErrors]	info[LsErrInfo]
MANCHESTER_ERR	Bit position in the word where error is injected. Zero is the most significant bit.
SYNCHRO_ERR	The desired sync. pattern. This is 6 bits each representing 0.5 uS of the sync. period. A '1' forces the sync. high, '0' forces it low. A pattern of 0 is not allowed.
WRD_LEN_ERR	The desired word length in bits. A value of 16 is the "normal" word length, 15 would be a word short by 1 bit.
RESP_TM_ERR	The required RT response time in microseconds.
POS_WRD_CNT_ERR	The number of extra words transmitted in the message.
NEG_WRD_CNT_ERR	The number of words to be omitted from the transmitted message.

info[LsErrPosn] contains the word number where the error will be injected when the info[LsErrors] is PARITY_ERR, SNCHRO_ERR or WRD_LEN_ERR. Word zero is the command or status word, depending on the setting of info[LsErrPhase].

F-HsRtRtIMsgGap

RT to RT Gap Time. info[F-HsRtRtIMsgGap] contains the gap time in uS between the two low speed parts (Command/Action pairs) of a high speed RT to RT transfer.

F_HsBus High Speed Bus. info[HsBus] contains the high speed bus to be used. This is either PRIMARY_BUS or SECONDARY_BUS.

F_HsErrors High Speed Error Injection. This should only be used when info[MsgType] specifies a high speed transfer. info[HsErrors] contains the type of error to be injected in the transfer. If it is set to NO_HS_ERRS then high speed error injection is disabled, otherwise one or a combination of high speed errors may be injected as follows:

PRE_BIT_CNT_ERR

The number of preamble bits, which is normally set to 40, is set to the value contained in info[HsErrInfo]

NEG_3910_WRD_CNT_ERR

The high speed frame is transmitted with one word less than normal.

POS_3910_WRD_CNT_ERR

The high speed frame is transmitted with one extra word appended.

FCS_ERR

The frame check sequence is transmitted with an invalid value.

HS_BIT_CNT_ERR

The number of bits to be removed from the HS data stream to create a bit count error. The number of bits to be removed is a value of 0-15 as defined in info[HsErrInfo]

HS_SD_ED_ERR

The contents of info[HsErrInfo] defines the bit pattern to be used for the start delimiter and end delimiter. The most significant byte defines the end delimiter and the least significant byte defines the start delimiter. For a good start and end delimiter this value shall be 0x8E71.

GATE_ERR

The single bit of the high speed frame is transmitted without a Manchester transition, such that it is low throughout the bit time. info[HsHErrPosn] and info[HsLErrPosn] specify the high and low order words of the bit position where this occurs. Bit 0 is the first bit of the preamble. If GATE_HIGH is OR'd with info[HsHErrPosn] then the bit will be transmitted high throughout the bit time.

F_HsRiTiTmOut

High Speed Receiver/Transmitter Initialise Time. Info[HsRiTiTmOut] is only used when info[MsgType] specifies a high speed transfer. This element should be set up as a Receiver Initialise Time or Transmitter Initialise Time as appropriate for the high speed transfer type.

msgData is a pointer to a DataInfo structure. Should msgData be NULL, then a default un initialised data buffer is allocated for the message. The size is sufficient to hold the number of words specified by info[WCnt] as appropriate based on the value of info[MsgType]

The DataInfo type is declared as:

```
typedef struct  { Sword  size;
                  Sword  posn;
                  Sword  action;
                  Uword  *data;
                  }  DataInfo;
```

size — Specifies the number of words to be modified.

posn — Is the offset in the data buffer where modification begins. A value of zero is the first data word. For high speed message data buffers the value -3 indicates the FCPA word, -2 the DA, and -1 the WC.

action — Specifies the action on the data buffer. This must be one of the following:

NEW — A new data buffer is created and filled with the data from the array pointed to by 'data'. In this case the value of 'posn' is ignored. 'size' words are copied from the data( ) into the new data buffer. The value stored at data (0) will be the first data word in the new data buffer.

OVERWRITE — The existing data starting at 'posn' and continuing for 'size' words is overwritten from the data in the array pointed to by 'data. The value at data(0) is copied to position 'posn' in the data buffer.

APPEND — The data in the array pointed to by 'data' is appended to the data buffer. In this case the value of 'posn' is ignored. 'size' words are copied.

INSERT — The data in the array pointed to by 'data', of length 'size' words is inserted into the data buffer at position 'posn', existing data being moved to make room for the insertion.

data — Is a pointer to an array of words.

Example;

```
#include "drivers.h"
…
Slong      msg1;
Uword      info1[SZ_MsgInfo};
DataInfo   datainfo;
Uword      data1[2];
Error      error;

/*
   Assuming the msg1 has been set to a low speed BC to RT message,
   change the word count to 5 and change words 2 and 3 to the values
   5555 and AAAA hex.
*/
info[0]             = F_WCnt;
info[WCnt]          = 5;

data1[0]            = 0x5555;
data1[1]            = 0xAAAA;

datainfo.size       = 2;
datainfo.posn       = 2;
datainfo.action     = OVERWRITE;
datainfo.data       = data1;

error = modMsg  (&card1, msg1, info1, &datainfo);
```

5.6. rdMsg( )

Error rdMsg (MemMapping *cardHandle, Slong msgId,
Uword *info, DataInfo, msgData)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
msgId	Identification number of an existing message, obtained from a call to crMsg[ ].
info	Pointer to an array of SZ_MsgInfo unsigned 16-bit quantities which will be filled in by rdMsg [ ] with the message information. See below.
msgData	Pointer to a DataInfo structure (shown below) which will be filled in by rdMsg [ ] with the data associated with the message.

Description

The rdMsg( ) function reads the contents of an existing message in the card identified by cardHandle. rdMsg uses the unique identifier assigned by a previous call to crMsg( ) to identify the message to be read. The info[ ] array holds the returned parameters. Should info be NULL then only data associated with the message will be read by rdMsg( ).

The elements of info[ ] which contain valid data are:

info[MsgType] Message Type. Info[MsgType] contains the type of message as follows:

BC_RT_3838	BC to RT low speed only.
RT_BC-3838	RT to BC low speed only.
RT_RT_3838	RT to RT low speed only.
B_BC_RT_3838	Bcast BC to RT low speed only.
B_RT_BC_3838	Bcast RT to BC low speed only.
B_BC_RT_CLK_3838	Bcast RX clock message.
TXMD_WODA_3838	3838 TX Mode no data.
TXMD_WIDA_3838	3838 TX Mode with data
RXMD_WIDA_3838	3838 RX Mode with data
B_MODE_WODA_3838	Bcast 3838 Mode no data
B_RXMD_WIDA_3838	Bcast 3838 Mode with data
BC_RT_3910	BC to RT with HS data.
RT_BC_3910	RT to BC with HS data.
RT_RT_3910	RT to RT with HS data.
B_BC_RT_3910	Bcast BC to RT with HS data.
B_RT_RT_3910	Bcast RT to RT with HS data.
TX_MSG_3910	RT to BC low speed from the RT's high speed sub-address (HS status/1st action word/bit words)
MODE_3910	HS Mode Command
B_MODE_3910	Bcast HS Mode Command.

info[LsIMsgGap] Inter Message Gap time, in multiples of 0.1 uS.

info[LsBus] Low speed Bus. This is either PRIMARY_BUS or SECONDARY_BUS.

info[RT1] RT Number of the RT involved in the message. For RT to RT transfers this is the RT that receives the data.

info[SubAdrs1] Sub Address. For low speed transfers, info[SubAdrs1] contains the sub-address of RT1 used in the transfer. For high speed transfers, info[SubAdrs1] contains the high speed identifier (sub-address) of the RT used in the transfer.

info[RT2] RT Number of the transmitting RT for RT to RT transfers, and is indeterminate for all other transfer types.

info[SubAdrs2] Sub Address. For low speed transfers, info[SubAdrs2] contains the sub-address of RT2 used in the transfer. For high speed transfers, info[SubAdrs2] contains the high speed identifier (sub-address) of the RT used in the transfer.

info[WCnt] Word count. Low speed or high speed word count depending on the type of message specified by info[MsgType].

info[ModeCode] Mode Code. Low speed or high speed Mode Code depending on the type of message specified by info[MsgType].

info[LsErrors] Low Speed Error. This will be one of the following:

NO_LS_ERRS	No LS Error.
PARITY_ERR	A parity error.
MANCHESTER_ERR	A Manchester error
SYNCHRO_ERR	A synchro. error
WRD_LEN_ERR	A word length error
WRONG_BUS_ERR	A wrong bus error
BOTH_BUS_ERR	Both bus error
RESP_TM_ERR	A response time error
POS_WRD_CNT_ERR	A +ve word count error
NEG_WRD_CNT_ERR	A -ve word count error

info[LsErrPhase] Low Speed Error Phase if info[LsErrors] is not NO_LS_ERRS. This will be one of the following:

ERR_DISABLED	No error injection
ERR_FIR_BC_TX	Error injected into command or data word transmitted by BC.
ERR_SEC_BC_TX	Error injected into second command or action word of a high speed RT to RT transfer.
ERR_FIR_RT_TX	Error injected into RT response message, status or data, when the card simulates the RT.
ERR_SEC_RT_TX	Error injected into second RT status word of an RT to RT transfer when the transmitting RT is simulated by the card.

info[LsErrPosn] Low Speed Error Position if info[LsErrors] is PARITY_ERR, SNCHRO_ERR or WRD_LEN_ERR. Word zero is the command or status word, depending on the setting of info[LsErrPhase].

info[LsErrInfo] Low Speed Error Information if info[LsErrors] is one of the following;

info[LsErrors]	info[LsErrInfo]
MANCHESTER_ERR	Bit position in the word where error is injected. Zero is the most significant bit.
SYNCHRO_ERR	The desired sync. pattern. This is 6 bits each representing 0.5 uS of the sync. period. A '1' forces the sync. high, '0' forces it low. A pattern of 0 is not allowed.
WRD_LEN_ERR	The desired word length in bits. A value of 16 is the "normal" word length, 15 would be a word short by 1 bit.
RESP_TM_ERR	The required RT response time in microseconds.
POS_WRD_CNT_ERR	The number of extra words transmitted in the message.
NEG_WRD_CNT_ERR	The number of words to be omitted from the transmitted message.

info[HsRtRtIMsgGap]
RT to RT Gap Time in uS when info[MsgType] specifies a high speed RT to RT transfer, else indeterminate.

info[HsBus] High Speed Bus when info[MsgType] specifies a high speed transfer, else indeterminate. This will be either PRIMARY_BUS or SECONDARY_BUS.

info[HsErrors] High Speed Error Injection if info[MsgType] specifies a high speed transfer. This will be set to NO_HS_ERRS if high speed error injection is disabled, otherwise it will be set to one or a combination of the following;

PRE_BIT_CNT_ERR
The number of preamble bits, which is normally set to 40, as set by the value contained in info[HsErrInfo]

NEG_3910_WRD_CNT_ERR
The high speed frame is transmitted with one word less than normal.

POS_3910_WRD_CNT_ERR
The high speed frame is transmitted with one extra word appended.

FCS_ERR
The frame check sequence is transmitted with an invalid value.

HS_BIT_CNT_ERR
The frame is transmitted with a bit count error.

HS_SD_ED_ERR
The frame is transmitted with an invalid START and/or END delimiter.

GATE_ERR

The single bit of the high speed frame is transmitted without a Manchester transition, such that it is low throughout the bit time. info[HsHErrPosn] and info[HsLErrPosn] specify the high and low order words of the bit position where this occurs. Bit 0 is the first bit of the preamble. If GATE_HIGH is OR'd with info[HsHErrPosn] then the bit will be transmitted high throughout the bit time.

info[HsErrInfo] The high speed preamble count, bit count error value or START/END delimiter pattern.

info[HsHErrPosn]
info[HsLErrPosn]

The position of the illegal Manchester bit if info[HsErrors] has the GATE_ERR set, else indeterminate. If the GATE_HIGH bit of info[HsHErrPosn] is set then a high level illegal bit will be transmitted, else a low level bit.

info[HsRiTiTmOut]

High Speed Receiver/Transmitter Initialise Time. info [HsRiTiTmOut] is only used when info[MsgType] specifies a high speed transfer

msgData is a pointer to a DataInfo structure. Should msgData be NULL then no data is read by rdMsg ( ).

The DataInfo type is declared as:

```
typedef struct   { Sword    size;
                   Sword    posn;
                   Sword    action;
                   Uword    *data;
                 } DataInfo;
```

size specifies the maximum number of words which will be written by rdMsg( ) into the data[ ] array.

posn is the offset in the data buffer where reading begins. A value of zero is the first data word. For high speed message data buffers the value -3 indicates the FCPA word, -2 the DA and -1 the WC.

action is unused.

data is a pointer to an array of words, into which rdMsg( ) copies the data associated with the message. This array must be at least 'size' words long. The first word in the ‘data’ array will be the value of the DDB Data Status Report for the message (DDB+6). The actual data will start at data[1].

Example:

```
#include "drivers.h"

…

Slong       msg1;
Uword       info1 [SZ_MsgInfo];
DataInfo    datainfo;
Sword       i;
Error       error

/*
Assuming that msg1 has been set up as a high speed message, read the   message parameters
and then the data associated with the message, then change the data using modMsg( )
*/
rdMsg (&card1, msg1, info1, NULL);  /* Read the message info  */

datainfo.size      = info1[WCnt];
datainfo.posn      = 0;

/*
Dynamically allocate a data buffer
*/
datainfo.data      = malloc (datainfo.size * sizeof(Uword));
error  =  rdMsg (&card1, msg1, NULL, &datainfo);

for (i=0; i< datainfo.size; i++)
            datainfo.data[i]= 0;

datainfo.action   = OVERWRITE;

/*
Release the dynamically allocated memory
*/
free (datainfo.data)
```

5.7. joinMsg( )

Error joinMsg (MemMapping *cardHandle, Slong Id1, Slong id2, Uword copyErr)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
id1, id2	id of previously created messages.
copyErr	Set if lookup table errors are to be the same.

Description

The joinMsg ( ) function points the lookup table of the message id2 to the same buffer as was created for the message id1.

Example:

```
#include "drivers.h"
…
Error     error;
Slong     id1, id2;
…
error  =  joinMsg ( &card1, id1, id2,  0);
```

5.8. linkMsg( )

Error linkMsg (MemMapping *cardHandle, Slong Id, LinkInfo *link)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
id	id of previously created message.
link	Pointer to LinkInfo structure defining the link buffers.

The elements of the LinkInfo structure are as follows:

mode	Mode of operation for link. This shall be one of the following values: LINK_BUFF_ALWAYS The option mask link bits are set for always. LINK_BUFF_ON_GOOD The option mask link bits are set for good msg. LINK_BUFF_WRITE Write to buffers that have already been created. LINK_BUFF_READ Read buffers that have already been created. LINK_BUFF_INIT Initialise the buffer pointer to bCount
bCount	This shall define the total number of buffers required, or in the case of LINK_BUFF_INIT, the buffer number to be pointed at in the lookup table.
BuffType	This shall define the buffer type as:- LS_SA - Low speed 3838 type buffer LS_MD - Low speed 3838 mode code buffer HS_SA - High speed 3910 type buffer
wCount	This shall define the number of words to read from the buffers or write to the buffers.
**buffer	This is an array of 'bCount' pointers. Each pointer must be set to point to a buffer of 'wCount' words or set to NO_PTR.

Description

Creating buffers:

- First create a message using crMsg() defining the data buffer size required.
- Now set mode for LINK_BUFF_ALWAYS or LINK_BUFF_ON_GOOD with the required number of buffers defined in 'bCount'.
- Now set 'buffers' to point to an array of 'bCount' pointers. Each pointer must point to a Uword array of data or set to the value NO_PTR. If the pointer is not NO_PTR then the data will be used to initialise the particular data buffer starting at the first data word. All buffer initialisation can be disabled by setting 'buffers' to NO_PTR.
- Set the buffer type and wCount and call linkMsg(). A further bCount-1 buffers will now be created with each link pointing to the next DDB. The last DDB will point back to the 1st DDB that was created using crMsg(). Each consecutive buffer, including the one that already existed, will be initialised with data values if the corresponding buffer[i] pointer is not set to NO_PTR.

Writing to buffers:

If further updating of buffers is required then this can be done by setting the mode to LINK_BUFF_WRITE and calling the function linkMsg(). The buffers pointed to by 'buffer' will be used to fill the data buffers on the card. If a buffer pointer is set to NO_PTR then the corresponding data buffer will not be written to.

Reading from buffers:

If a data buffer(s) is required to be read then set the corresponding buffer[i] pointer to a Uword array for storing the data. If you do not wish to read a particular buffer then set the pointer to NO_PTR. Now call linkMsg with mode set to LINK_BUFF_READ. The read back includes header information as follows:

LS Buffer: DDB Status	HS Buffer: DDB Status
TTAG High	TTAG High
TTAG Low	TTAG Low
Data 1	FC,PA
|	DA
Data n	WC
	Data 1
	|
	Data n

Note 1:
The order of the buffers will be in the order in which they are linked. The 1st buffer, corresponding to buffer[0], will be the buffer that was initially created using crMsg().

Note 2:
The LabVIEW version only allows the writing and reading of 1 buffer when in LINK_BUFF_READ and LINK_BUFF_WRITE only. A maximum of 20 buffers are allowed.

Initialising the lookup buffer pointer:

To initialise the lookup buffer, set the 'bCount value to the buffer number. The buffer numbers start at 1. Now set the mode to LINK_BUFF_INIT and execute the function.

Example:

```
#include "drivers.h"
…
Error      error;
LinkInfo   link;
Slong      id;
…
…

link.mode         = LINK_BUFF_ALWAYS;
link.bCount       = 8;
link.buffType     = LS_SA;
link.wCount       = 32;
link.buffer       = NO_PTR;

error  =  linkMsg( &card1, id, &link);
```

5.9. insertMsg( )

Error insertMsg (MemMapping *cardHandle, InsertInfo *insertData)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
*insertData	Pointer to InsertInfo structure defining the insertion parameters.

The elements of the InsertInfo structure are as follows:

nmrMsgs	Number of message in message list to be inserted.
*msgIdL	Pointer to a list of message id values to be inserted. The list size must be the same as the value of nmrMsgs.

Description

Acyclic messages can be inserted into the frame as follows:

- Create the messages you wish to insert using the standard crMsg().
- Set the number of messages you wish to insert in nmrMsgs.
- Set the list of the messages in the order in which you wish them to be transmitted.
- Call the function insertMsg().

Example:

```
#include "drivers.h"
…
Error       error;
InsertInfo  insertData;
Slong       id1, id2, id3, list[3];
…
…
list[0] = id1;
list[1] = id2;
list[2] = id3;
insertData.msgIdL    = list;
insertData.nmrMsgs = 3;

error = insertMsg(&card1, &insertData);
```

5.10. crCycle( )

Error crCycle (MemMapping *cardHandle, Slong *cycleId, CycleInfo *cycleData)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
cycleId	Pointer to a variable which crCycle ( ) fills in with the identifier of the cycle created.
cycleData	Pointer to a CycleInfo structure (shown below) which defines the cycle to be created.

Description

The crCycle ( ) function creates a new cycle in the card identified by cardHandle. A cycle is a sequence of messages to be transmitted in a fixed time. This time is set by the call to setup( ) as the BcCycleTm. The transmission of a cycle always begins on a multiple of the time period.

The cycle created by crCycle is part of a frame (see crFrame ( ) function). A cycle can be thought of as a "minor cycle" and a frame as a "major cycle".

The CycleInfo type is declared as:

```
typedef struct   { Sword  nmrMsgs;
                   Slong   *msgIdL;
                   Sword   action;
                   Sword   posn;
                 } CycleInfo;
```

nmrMsgs	is the number of message identifiers in the cycle list.
msgIdL	points to a list of message identifiers, created by calls to crMsg( ). This list must be 'nmrMsgs' in length. The list of messages will be transmitted in the sequence specified.
action	specifies the action on the cycle. This must be set to NEW in order to create a cycle.
posn	is unused

Example:

```
#include "drivers.h"

…

Slong       msgList[4];
CycleInfo   cycledef;
Slong       cycle1;
Error       error

/*
Assign msgList[ ] to 4 messages by calling crMsg[ ]
(Not shown here]
*/
cycledef.nmrMsgs        =  4;
cycledef.msgIdL         = msgList;
cycledef.action         = NEW;

error  =  crCycle ( &card1, &cycle1, &cycledef);
```

5.11. modCycle( )

Error modCycle (MemMapping *cardHandle, Slong cycleId, CycleInfo *cycleData)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
cycleId	The identification number of an existing cycle, obtained from a call to crCycle ( ).
cycleData	Pointer to a CycleInfo structure (shown below) which defines how the cycle is to be modified.

Description

The modCycle ( ) function modifies an existing cycle in the card identified by cardHandle. modCycle ( ) uses the unique identifier assigned by a previous call to crCycle ( ) to identify the cycle to be modified.

The CycleInfo type is declared as:

```
typedef struct   { Sword  nmrMsgs;
                   Slong   *msgIdL;
                   Sword   action;
                   Sword   posn;
                 } CycleInfo;
```

nmrMsgs — is the number of message identifiers in the cycle to be modified.

msgIdL — points to a list of message identifiers, created by calls to crMsg().
This list must be 'nmrMsgs' in length. The list of messages will be copied into the existing cycle.

action — specifies the action on the cycle. This must be one of the following:

NEW	The list of message identifiers, of length 'nmrMsgs' replaces the existing message list. 'posn' is unused and the resulting cycle has exactly 'nmrMsgs' in it.
OVERWRITE	The existing sequence of message identifiers staring at 'posn' and continuing for 'nmrMsgs' is overwritten from the message list 'msgIdL'. The cycle is expanded if necessary.
APPEND	The message is appended to the existing cycle.
INSERT	The message list of length 'nmrMsgs' is inserted into the existing cycle at position 'posn'. Existing message identifiers are moved to make room for the insertion.

posn — is unused

Example:

```
#include "drivers.h"

…

Slong       msgList[2];
CycleInfo   cycledef;
Slong       cycle1;
Error       error

/*
Insert msgList[ ] of 2 messages into the cycle.  The messages must have
previously been created using crMsg[ ], and the cycle by crCycle( )
*/
cycledef.nmrMsgs        =  2;
cycledef.msgIdL         = msgList;
cycledef.action         = INSERT;
cycledef.posn           = 1;
error  =  modCycle ( &card1, &cycle1, &cycledef);
```

5.12. rdCycle( )

Error rdCycle (MemMapping *cardHandle, Slong cycleId, CycleInfo *cycleData)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
cycleId	The identification number of an existing cycle, obtained from a call to crCycle ( ).
cycleData	Pointer to a CycleInfo structure (shown below) which will be filled in by rdCycle( ) with the cycle definition.

Description

The rdCycle ( ) function reads the contents of an existing cycle in the card identified by cardHandle. rdCycle ( ) uses the unique identifier assigned by a previous call to crCycle ( ) to identify the cycle to be read.

The CycleInfo type is declared as:

```
typedef struct   { Sword  nmrMsgs;
                   Slong   *msgIdL;
                   Sword   action;
                   Sword   posn;
                 } CycleInfo;
```

nmrMsgs	specifies the maximum number of message identifiers which will be written by rdCycle( ) into the msgIdL( ) array. rdCycle will then set this value to the number actually read, which will be less than or equal to the value prior to reading the function in the cycle to be modified.
msgIdL	points to a list of message identifiers, filled in by rdCycle with the message identifiers that comprise the cycle.
action	not used.
posn	specifies the position in the cycle of the first identifier to be read.

Example:

```
#include "drivers.h"

…

Slong       msglist[32];
CycleInfo   cycledef;
Slong       cycle1;
Error       error

/*
Read all the message identifiers of the cycle.  Limit the list to 32 maximum.
*/
cycledef.nmrMsgs        =  32;
cycledef.msgIdL         = msgList;
cycledef.posn           = 0             /* start at first message  */

error  =  rdCycle ( &card1, &cycle1, &cycledef);
```

5.13. crFrame( )

Error crFrame (MemMapping *cardHandle, Slong *frameId, FrameInfo *frameData)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
frameId	Pointer to a variable which crFrame( ) fills in with the identifier of the frame created.
frameData	Pointer to a FrameInfo structure (shown below) which defines the frame to be created.

Description

The crFrame ( ) function creates a new frame in the card identified by cardHandle. A frame is a sequence of cycles to be transmitted in a sequence. Each cycle is transmitted in a fixed time (set by a call to setup ( ) as the BcCycleTm), resulting in a fixed frame time equivalent to the cycle time multiplied by the number of cycles to be transmitted.

A cycle can be thought of as a "minor cycle" and a frame as a "major cycle".

The FrameInfo type is declared as:

```
typedef struct   { Sword  nmrCycles;
                   Slong   *cyIdL;
                   Sword   action;
                   Sword   posn;
                 } FrameInfo;
```

nmrCycles	is the number of cycle identifiers in the frame to be created.
cyIdL	points to a list of cycle identifiers, created by calls to crCycle( ). This list must be 'nmrCycles' in length. The list of cycles will be transmitted in the sequence specified.
action	specifies the action on the frame. This must be set to NEW in order to create a frame.
posn	is unused.

Example:

```
#include "drivers.h"
…

Slong        cyclelist[8];
FrameInfo    framedef;
Slong        frame1;
Error        error

/*
Assign cyclelist[ ] to 8 cycles by calling crCycle( )
*/
framedef.nmrCycles      =  8;
framedef.cyIdL          = cyclelist;
framedef.action         = NEW;
error  =  crFrame ( &card1, &frame1, &framedef);
```

5.14. crFrame( )

Error modFrame (MemMapping *cardHandle, Slong frameId, FrameInfo *frameData)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
frameId	Identification number of an existing frame, obtained by call to crFrame( ).
frameData	Pointer to a FrameInfo structure (shown below) which defines how the frame is to be modified.

Description:

The modFrame ( ) function modifies an existing frame in the card identified by cardHandle. modFrame ( ) uses the unique identifier assigned by a previous call to crFrame( ) to identify the frame to be modified.

The FrameInfo type is declared as:

```
typedef struct   { Sword  nmrCycles;
                   Slong   *cyIdL;
                   Sword   action;
                   Sword   posn;
                 } FrameInfo;
```

nmrCycles — is the number of cycle identifiers in the frame to be modified.

cyIdL — points to a list of cycle identifiers, created by calls to crCycle( ). This list must be 'nmrCycles' in length. The list of cycles will be copied into the existing frame.

action — specifies the action on the frame, and must be one of the following:

NEW	The list of message identifiers, of length 'nmrMsgs' replaces the existing cycle list. 'posn' is unused and the resulting frame has exactly 'nmrCycles' in it.
OVERWRITE	The existing sequence of cycle identifiers staring at 'posn' and continuing for 'nmrCycles' is overwritten from the cycle list 'cyIdL'. The frame is expanded if necessary.
APPEND	The cycle list is appended to the existing frame. 'posn' is unused.
INSERT	The cycle list of length 'nmrCycles' is inserted into the existing frame at position 'posn'. Existing cycle identifiers are moved to make room for the insertion.

Example:

```
#include "drivers.h"
…

Slong       cyclelist[1];
FrameInfo   framedef;
Slong       frame1;
Error       error

/*
Change the second cycle of the frame to a new value, created by a call to crCycle( )
*/
framedef.nmrCycles      =  1
framedef.cyIdL          =  cyclelist;
framedef.action         =  OVERWRITE;
framedef.posn           =  1;
error  =  crFrame ( &card1, &frame1, &framedef);
```

5.15. rdFrame( )

Error rdFrame (MemMapping *cardHandle, Slong frameId, FrameInfo *frameData)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
frameId	The identification number of an existing frame, obtained from a call to crFrame ( ).
frameData	Pointer to a FrameInfo structure (shown below) which will be filled in by rdFrame( ) with the frame definition.

Description

The rdFrame ( ) function reads the contents of an existing frame in the card identified by cardHandle. rdFrame ( ) uses the unique identifier assigned by a previous call to crFrame ( ) to identify the frame to be read.

The FrameInfo type is declared as:

```
typedef struct   { Sword  nmrCycles;
                   Slong   *cyIdL;
                   Sword   action;
                   Sword   posn;
                 } FrameInfo;
```

nmrCycles	specifies the maximum number of cycle identifiers which will be written by rdFrame( ) into the cyIdL( ) array. rdFrame will then set this value to the number actually read, which will be less than or equal to the value prior to calling the function.
cyIdL	points to a list of cycle identifiers, filled in by rdFrame with the cycle identifiers that comprise the frame.
action	not used.
posn	specifies the position in the frame of the first identifier to be read.

Example:

```
#include "drivers.h"

…

Slong        cyclelist[64];
FrameInfo    framedef;
Slong        frame1;
Error        error

/*
Read all the cycle identifiers of the frame..  Limit the list to 64 maximum.
*/
framedef.nmrCycles      =  64;
framedef.cyIdL          = cyclelist;
cycledef.posn           = 0                 /* start at first cycle  */
error  =  rdCycle ( &card1, &cycle1, &cycledef);
```

5.16. rdFrame( )

Error runFrame (MemMapping *cardHandle, Slong frameId, Uword count)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
frameId	The identification number of an existing frame, obtained from a call to crFrame ( ).
count	Number of times that the specified frame shall be run.

Description:

The runFrame ( ) function initiates the transmission of the specified frame from the card identified by cardHandle.

frameId is a signed 32-bit quantity which specifies the ID of the frame to be transmitted. This parameter is mandatory.

count is an unsigned 16-bit quantity which specifies the number of times the frame is to be transmitted. A value of zero for count will cause the frame to be transmitted indefinitely until the haltFrame( ) driver is called.

Example:

```
#include "drivers.h"

…

Slong       frame1;
Error       error
…

error  =  runFrame ( &card1, frame1, 10);
```

5.17. haltFrame( )

Error haltFrame (MemMapping *cardHandle)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).

Description

The haltFrame ( ) function halts the transmission of the currently transmitting frame from the card identified by cardHandle.

Example:

```
#include "drivers.h"

…

Error        error
…

error = haltFrame (&card1);
```

6. REMOTE TERMINAL FUNCTIONS

6.1. INTRODUCTION

The Remote Terminal functions manage the setup of the card and control the transmission of data when in MRT mode. The drivers set-up all the RT's to default values during setup( ). Thus all RT's, Sub-addresses and mode codes are defined before the RT drivers are called. These drivers assign unique sub-addresses, data and errors for chosen RT/SA or RT/MODE CODE pairs.

The RT's default as follows:

RtState	DISABLED_RT
RtLsLastCmd	0
RtStatus	RT 0 0; RT 1 800 hex; RT 2 1000 hex etc.
RtVector	0
RtLsBitWord	0
RtHsStatus	0
RtHsLastAct	0
RtHsBitWord	0
Global Errors	DISABLED
HS mode codes	TX AND RX ENABLED

The low speed sub-addresses:

No error Injection

All RX sub-addresses point to one data buffer	(32 data words long)
All TX sub-addresses point to another data buffer	(32 data words long)

The high speed sub-addresses:

No error Injection

All RX sub-addresses point to one data buffer	(4104 data words long)
All TX sub-addresses point to another data buffer	(4104 data words long)

The low speed Mode Codes:

No error Injection

All RX Mode codes point to one data buffer	(32 data words long)
All TX Mode codes point to another data buffer	(32 data words long)

The modMrtRt( ) driver must be called to set-up all simulated RT's before the crMrtRtSa() driver is called.

.

6.2. modMrtRt( )

Error modMrtRt (MemMapping *cardHandle, Sword rtNum, Uword *info)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
rtNum	The number of the RT to be modified.
info	Pointer to an array of size SZ_MrtRtInfo unsigned 16-bit quantities which modify an RT. See below.

Description

The modMrtRt ( ) function modifies an RT in MRT mode in the card identified by cardHandle. It is essential that this has been initialised via a previous call to setup( ). The MRT's automatically default as follows during setup( ).

Element	Default value
RtState	DISABLED_RT
RtStatus	RT 0 0; RT 1 800 hex; RT 2 1000 hex etc.
RtVector	0
RtLsBitWord	0
RtHsBitWord	0
RtGlobLsErrors	NO_LS_ERRS
RtLsErrBusDis	NO_BUSES
RtHsModesDis	NONE

rtNum is a signed 16-bit quantity. It is the RT which is to be modified and it must be set by the user. It must contain a valid RT number. This parameter is MANDATORY.

info[ ] defines the modification data for the RT. The first element is a mask organised on a bit basis defining which elements of the info[ ] array are present. The fields of info[0], which may be OR'd together are:-

F_RtState	RT State. info[RtState] contains the simulation state of the card. It must be one of the following:- SIMULATED_RT The RT is simulated by the card (An INTERNAL RT) DISABLED_RT The RT is off-card (An EXTERNAL RT)
F_RtStatus	RT Status. info[RtStatus] contains the status word which will be transmitted by the RT when the RT is simulated.
F_RtVector	RT Vector Word. info[RtVector] contains the vector word which will be transmitted by the RT when the Transmit Vector Word mode code command is received (and the RT is simulated).

F_RtLsBitWord — RT LS BIT Word. info[RtLsBitWord] contains the LS BIT word which will be transmitted by the RT when the Transmit Bit Word mode code command is received (and the RT is simulated).

F_RtHsBitWord — RT HS BIT Word. info[RtHsBitWord] contains the HS BIT word which will be transmitted by the RT when a transmit command is sent to the HS Sub Address (and the RT is simulated).

F_RtGlobLsErrors — Define Global RT Error. This low speed error is common to all low speed sub-addresses on the card. info[RtGlobLsErrors] must be one of the following:

NO_3838_RESP_ERR	No response from RT
NO_LS_ERRS	Low speed error injection disabled
PARITY_ERR	Inject a parity error
MANCHESTER_ERROR	Inject a Manchester error
SYNCRO_ERR	Inject a synchro. Error
WRD_LEN_ERR	Inject word length error
WRONG_BUS_ERR	Inject wrong bus error
BOTH_BUS_ERR	Transmit on both buses
RESP_TM_ERR	Inject a response time error
POS_WRD_CNT_ERR	Inject a positive word count error
NEG_WRD_CNT_ERR	Inject a negative word count error

info[RtGlobLsErrInfo] contains extra low speed error information which depends on the value of info[RtGlobLsErrors].

info[RtGlobLsErrors]	info[RtGlobLsErrInfo]
MANCHESTER_ERR	Bit position in the word where error is injected. Zero is the most significant bit.
SYNCHRO_ERR	The desired sync. pattern. This is 6 bits each representing 0.5 uS of the sync. period. a '1' forces the sync. high, '0' forces it low. A pattern of zeros is not allowed.
WRD_LEN_ERR	The desired word length in bits. A value of 16 is the 'normal' word length, 15 is a word short by 1 bit.
RESP_TM_ERR	The required RT response time in microseconds.
POS_WRD_CNT_ERR	The number of extra words to be transmitted.
NEG_WRD_CNT_ERR	The number of words that will be omitted from the message.

info[RtGlobLsErrPosn] contains the word number where the error will be injected when info[RtGlobLsErrors] is PARITY_ERR, SYNCRO_ERR or WRD_LEN_ERR. Word zero is the status word, word 1 the first data word, etc.

F_RtLsErrBusDis — Disable Error Bus. info[RtLsErrBusDis] contains the bus(es) which can not have errors injected. Permitted values are:

PRIMARY_BUS	Disabled on primary bus
SECONDARY_BUS	Disabled on secondary bus
BOTH_BUSES	Disabled on both buses
NO_BUSES	Enabled on both buses

F_RtHsModesDis — Disable high speed Mode codes. info[RtHsModesDis] defines which high speed mode codes are disabled for this RT. . Permitted values are:

RXTYPE	All receive mode codes disabled
TXTYPE	All transmit mode codes disabled
BOTH	All mode codes disabled
NONE	All mode codes enabled.

Example:

```
#include "drivers.h"

…

Uword   info[SZ_MrtRtInfo];
Sword   rtNum;
Error     error;

/*
Modify RT 1 such that it simulates (i.e. on card RT), high speed mode codes disabled and all
low speed sub-addresses have a Manchester error on bit1 of the status word.
*/
rtNum = 1

info[WhatToSet]          = F_RtState ¦ F_RtHsModesDis ¦ F_RtGlobLsErrors;
info[RtState]            = SIMULATED_RT;
info[RtHsModesDis]       = BOTH;
info[RtGlobLsErrors]     = MANCHESTER_ERR;
info[RtGlobLsErrInfo]    = 1;
info[RtGlobLsErrPosn]    = 0;
error = modMrtRt  (&card1, rtNum, info);
```

6.3. rdMrtRt( )

Error rdMrtRt (MemMapping *cardHandle, Sword rtNum, Uword *info)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
rtNum	The number of the RT to be read.
info	Pointer to an array of size SZ_MrtRtInfo unsigned 16-bit quantities which modify an RT.

Description

The rdMrtRt ( ) function reads an RT in MRT mode in the card identified by cardHandle. It is essential that this has been initialised via a previous call to setup( ). rtNum is a signed 16-bit quantity. It is the RT which is to be modified and it must be set by the user. It must contain a valid RT number. This parameter is MANDATORY. info[ ] is filled in by rdMrtRt ( ) with values read from the card related to RT rtNum. If info is NULL then no data is read. The fields are unused in rdMrtRt ( ). The elements of info [ ] which contain data are:

RtState	RT State. info[RtState] will contain the simulation state of the card. It will be set to one of the following:- SIMULATED_RT The RT is simulated by the card (An INTERNAL RT) DISABLED_RT The RT is off-card (An EXTERNAL RT)
RtStatus	RT Status. info[RtStatus] contains the status word being transmitted by the RT when the RT is simulated.
RtVector	RT Vector Word. info[RtVector] contains the vector word being transmitted by the RT when the Transmit Vector Word mode code command is received (and the RT is simulated).
RtLsBitWord	RT LS BIT Word. info[RtLsBitWord] contains the LS BIT word being transmitted by the RT when the Transmit Bit Word mode code command is received (and the RT is simulated).
RtHsBitWord	RT HS BIT Word. info[RtHsBitWord] contains the HS BIT word being transmitted by the RT when a transmit command is sent to the HS Sub Address (and the RT is simulated).

RtGlobLsErrors — Define Global RT Error. This low speed error is common to all low speed sub-addresses on the card. info[RtGlobLsErrors] will be one of the following:

NO_3838_RESP_ERR	No response from the RT
NO_LS_ERRS	Low speed error injection disabled
PARITY_ERR	Inject a parity error
MANCHESTER_ERROR	Inject a Manchester error
SYNCRO_ERR	Inject a synchro. error
WRD_LEN_ERR	Inject word length error
WRONG_BUS_ERR	Inject wrong bus error
BOTH_BUS_ERR	Transmit on both buses
RESP_TM_ERR	Inject a response time error
POS_WRD_CNT_ERR	Inject a positive word count error
NEG_WRD_CNT_ERR	Inject a negative word count error

info[RtGlobLsErrInfo] contains extra low speed error information which depends on the value of info[RtGlobLsErrors].

info[RtGlobLsErrors].	info[RtGlobLsErrInfo]
MANCHESTER_ERR	Bit position in the word where error is injected. Zero is the most significant bit.
SYNCHRO_ERR	The desired sync. pattern. This is 6 bits each representing 0.5 uS of the sync. period. a '1' forces the sync. high, '0' forces it low. A pattern of zeros is not allowed.
WRD_LEN_ERR	The desired word length in bits. A value of 16 is the 'normal' word length, 15 is a word short by 1 bit.
RESP_TM_ERR	The required RT response time in microseconds.
POS_WRD_CNT_ERR	The number of extra words to be transmitted.
NEG_WRD_CNT_ERR	The number of words that will be omitted from the message.

info[RtGlobLsErrPosn] contains the word number where the error will be injected when info[RtGlobLsErrors] is PARITY_ERR, SYNCRO_ERR or WRD_LEN_ERR. Word zero is the status word, word 1 the first data word, etc.

RtLsErrBusDis — Disable Error Bus. info[RtLsErrBusDis] contains the bus(es) which can not have errors injected. This will be one of the following:

PRIMARY_BUS	Disabled on primary bus only.
SECONDARY_BUS	Disabled on secondary bus only.
BOTH_BUSES	Disabled on both buses.
NO_BUSES	Enabled on both buses.

RtHsModesDis	Disable high speed Mode codes. info[RtHsModesDis] contains the state of the high speed mode codes, and will be one of the following:

RXTYPE	All receive mode codes disabled
TXTYPE	All transmit mode codes disabled
BOTH	All mode codes disabled
NONE	All mode codes enabled.

RtLsLastCmd	Last low speed command word. info[RtLsLastCmd] contains the last command word received by the RT when in simulate mode.
RtHsStatus	High speed Status word. info[RtHsStatus] contains the RT high speed status word when in simulate mode.
RtHsLastAct	High speed action word.. info[RtHsLastAct] contains the RT last action word when in simulate mode.

Example:

```
#include "drivers.h"

…

Uword   info[SZ_MrtRtInfo];
Sword   rtNum;
Error   error;

/*
Read from RT 1.
*/

rtNum = 1

error = rdMrtRt  (&card1, rtNum, info);
```

6.4. crMrtRtSa( )

Error crMrtRtSa(MemMapping *cardHandle, Sword rtNum,
SaInfo *sa, Sword saType)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
rtNum	The number of the RT to be configured.
sa	Pointer to a structure of type SaInfo which contains information related to the sub-address / mode code.
saType	The type of sub-address. Permitted values are: LS_SA Low speed sub-address HS_SA High speed sub-address

Description

The crMrtRtSa ( ) function creates unique data buffers and error injection for the given RT/SA pair on the card identified by cardHandle. It is essential that this has been initialised via a previous call to setup( ).

sa is a pointer to a SaInfo structure. If sa is NULL then this is an error. The SaInfo type is defined as:

```
typedef struct {  Sword     sa;
                  Sword     wrap;
                  Sword     rxWCnt;
                  Sword     txWCnt;
                  Sword     hsRiTm;
                  Sword     hsTiTm;
                  DataInfo  *rxData;
                  DataInfo  *txData;
                  ErrorInfo  *rxErrData;
                  ErrorInfo  *txErrData;
               }  SaInfo;
```

sa	Sub-address number. sa contains the Low Speed sub-address, or High Speed sub-address number. This parameter is MANDATORY and must be set to a valid sub-address number.
wrap	Wrap type. wrap contains the wrap type. This specifies whether the RX and TX data buffers are common or not. It must be set to one of the following values:- WRAP RX and TX have the same data buffer NO_WRAP RX and TX have different data buffers If omitted, crMrtRtSa( ) uses NO_WRAP as the default value.

rxWCnt — Receive Word-count. For Low Speed transfers rxWCnt contains the number of words to be received. For High Speed transfers rxWCnt contains the number of words in the INFO field of the HS frame to be received. This parameter is mandatory. To use the default value (32) set it to DEFVAL.

txWCnt — Transmit Word-count. For Low Speed transfers txWCnt contains the number of words to be transmitted. For High Speed transfers txWCnt contains the number of words in the INFO field of the HS frame to be transmitted. This parameter is mandatory. To use the default value (32) set it to DEFVAL

hsRiTm — HS Receiver Initialisation Timeout. hsRiTm contains the High Speed Receiver Initialisation Time in uS. This parameter is mandatory. To use the default value (200uS) set it to DEFVAL

hsTiTm — HS Transmitter Initialise Time. hsTiTm contains the High Speed Transmitter Initialise Time in uS. This parameter is mandatory. To use the default value (24uS) set it to DEFVAL

rxData — RX Data. rxData contains a pointer to a DataInfo structure which defines the data associated with this RX sub-address. Should rxData be NULL then a default un-initialised data buffer is allocated for the sub-address. The size is sufficient to hold the number of words specified by rxWCnt. If omitted, crMrtRtSa( ) uses NULL as the default value.

The DataInfo type is declared as :-

```
typedef struct   { Sword   size ;
                   Sword   posn ;
                   Sword   action ;
                   Uword   *data ;
                 } DataInfo ;
```

size specifies the size in words of the data buffer to be created. crMrtRtSa( ) automatically accounts for the extra header words in a High Speed message. A value of 0 indicates that the default size should be used as if rxData was NULL.

posn is unused.

action specifies the action on the data buffer - this must be set to NEW in order to create a new data buffer.

data is a pointer to an array of words which are copied into the newly created data buffer. If data is NULL then the data buffer contents are indeterminate. At most 'size' words are copied into the data buffer.

txData TX Data. txData contains a pointer to a DataInfo structure which defines the data associated with this TX sub-address. Should txData be NULL then a default un-initialised data buffer is allocated for the sub-address. The size is sufficient to hold the number of words specified by txWCnt. If omitted, crMrtRtSa( ) uses NULL as the default value.

The DataInfo type is declared as :-

```
typedef struct    { Sword   size ;
                    Sword   posn ;
                    Sword   action ;
                    Uword   *data ;
                  }  DataInfo ;
```

size specifies the size in words of the data buffer to be created. crMrtRtSa( ) automatically accounts for the extra header words in a High Speed message. A value of 0 indicates that the default size should be used as if rxData was NULL.

posn is unused.

action specifies the action on the data buffer - this must be set to NEW in order to create a new data buffer.

data is a pointer to an array of words which are copied into the newly created data buffer. If data is NULL then the data buffer contents are indeterminate. At most 'size' words are copied into the data buffer.

rxErrData RX Error Data. rxErrData contains a pointer to an ErrInfo structure which defines the error associated with this RX sub-address. Should rxErrData be NULL then there is no error. If omitted, crMrtRtSa( ) uses NULL as the default value.

The ErrInfo type is declared as :-

```
typedef struct    { Uword   lsErrors;
                    Uword   lsErrInfo;
                    Uword   lsErrPosn;
                    Uword   hsErrors;
                    Uword   hsErrInfo;
                    Uword   hsHErrPosn;
                    Uword   hsLErrPosn;
                  } ErrInfo;
```

lsErrors contains the type of error to be injected in the Low Speed transfer. This must be one of the following:-

NO_LS_ERRS	Low Speed error injection disabled
PARITY_ERR	Inject a parity error
MANCHESTER_ERR	Inject a Manchester error
SYNCHRO_ERR	Inject a synchro. error
WRD_LEN_ERR	Inject a word length error
WRONG_BUS_ERR	Inject a wrong bus error
BOTH_BUS_ERR	Transmit on both buses
RESP_TM_ERR	Inject a response time error

If omitted, crMrtRtSa( ) uses NO_LS_ERRS as the default value.

lsErrInfo contains extra Low Speed error information which depends on the value of lsErrors.

lsErrors	lsErrInfo
MANCHESTER_ERR	Bit Position in the word where error is injected. 0 is the most significant bit.
SYNCHRO_ERR	The desired sync pattern. This is 6 bits each representing 0.5 uS of the sync period. A '1' forces the sync high, '0' forces it low. A pattern of 0 is not allowed.
WORD_LEN_ERR	The desired word length in bits. A value of 16 is the "normal" word length, 15 is a word short by 1 bit.
RESP_TM_ERR	The required RT response time in microseconds.
POS_WRD_CNT_ERR	The number of extra words transmitted in the message.
NEG_WRD_CNT_ERR	The number of words that will be omitted from the message.

lsErrPosn contains the word number where the error will be injected when lsErrors is PARITY_ERR, SYNCHRO_ERR or WRD_LEN_ERR. For an RX message an error can only be injected into the status word. Thus lsErrPosn must be set to zero.

hsErrors contains the type of error to be injected in the High Speed transfer. This will be NO_HS_ERRS or a combination of the following:-

PRE_BIT_CNT_ERR	The number of pre-amble bits which is normally set to 40 is set to the value contained in hsErrInfo
NEG_3910_WRD_CNT_ERR	The High Speed Frame is transmitted with one word less than normal.
POS_3910_WRD_CNT_ERR	The High Speed Frame is transmitted with one extra word appended.
FCS_ERR	The Frame Check Sequence is transmitted with an invalid value.

HS_BIT_CNT_ERR

The number of bits to be removed from the HS data stream to create a bit count error. The number of bits to be removed is a value of 0-15 as defined in info[HsErrInfo]

HS_SD_ED_ERR

The contents of info[HsErrInfo] defines the bit pattern to be used for the start delimiter and end delimiter. The most significant byte defines the end delimiter and the least significant byte defines the start delimiter. For a good start and end delimiter this value shall be 0x8E71.

GATE_ERR	A single bit of the High Speed frame is transmitted without a Manchester Transition such that it is low throughout the bit time. hsLErrPosn and hsHErrPosn specify the high and low order words of the bit position where this occurs. Bit 0 is the first bit of the pre-amble. If GATE-HIGH is OR'd with hsHErrPosn then the bit will be transmitted high throughout the bit time.

If omitted, crMrtRtSa( ) uses NO_HS_ERRS as the default value.

NOTE: When defining errors you may define either a LS or HS error to be associated with the sub-address. If defining LS error then hsErrors must be set to NO_HS_ERRORS. If defining HS error then lsErrors must be set to NO_LS_ERRORS.

txErrData — TX Error Data. txErrData contains a pointer to an ErrInfo structure which defines the error associated with this TX sub address. Should txErrData be NULL then there is no error. If omitted, crMrtRtSa( ) uses NULL as the default value.

The ErrInfo type is declared as :-

```
typedef struct {  Uword    lsErrors;
                  Uword   lsErrInfo;
                  Uword   lsErrPosn;
                  Uword   hsErrors;
                  Uword   hsErrInfo;
                  Uword   hsHErrPosn;
                  Uword   hsLErrPosn;
                } ErrInfo;
```

lsErrors contains the type of Low Speed error to be injected in the transfer. This must be one of the following:-

NO_LS_ERRS	Low Speed error injection disabled
PARITY_ERR	Inject a parity error
MANCHESTER_ERR	Inject a Manchester error
SYNCHRO_ERR	Inject a synchro. error
WRD_LEN_ERR	Inject a word length error
WRONG_BUS_ERR	Inject a wrong bus error
BOTH_BUS_ERR	Transmit on both buses
RESP_TM_ERR	Inject a response time error
POS_WRD_CNT_ERR	Inject a positive word count error
NEG_WRD_CNT_ERR	Inject a negative word count error

lsErrInfo contains extra Low Speed error information which depends on the value of lsErrors.

lsErrors	lsErrInfo
MANCHESTER_ERR	Bit Position in the word where error is injected. 0 is the most significant bit.
SYNCHRO_ERR	The desired sync pattern. This is 6 bits each representing 0.5 uS of the sync period. A '1' forces the sync high, '0' forces it low. A pattern of 0 is not allowed.
WORD_LEN_ERR	The desired word length in bits. A value of 16 is the "normal" word length, 15 is a word short by 1 bit.
RESP_TM_ERR	The required RT response time in microseconds.
POS_WRD_CNT_ERR	The number of extra words transmitted in the message.
NEG_WRD_CNT_ERR	The number of words that will be omitted from the message.

lsErrPosn contains the word number where the error will be injected when lsErrors is PARITY_ERR, SYNCHRO_ERR or WRD_LEN_ERR. Word zero is the status word, word 1 the first data word, etc.

hsErrors contains the type of error to be injected in the High Speed transfer. This will be NO_HS_ERRS or a combination of the following:-

PRE_BIT_CNT_ERR	The number of pre-amble bits which is normally set to 40 is set to the value contained in hsErrInfo
NEG_3910_WRD_CNT_ERR	The High Speed Frame is transmitted with one word less than normal.
POS_3910_WRD_CNT_ERR	The High Speed Frame is transmitted with one extra word appended.
FCS_ERR	The Frame Check Sequence is transmitted with an invalid value.

HS_BIT_CNT_ERR

The number of bits to be removed from the HS data stream to create a bit count error. The number of bits to be removed is a value of 0-15 as defined in info[HsErrInfo]

HS_SD_ED_ERR

The contents of info[HsErrInfo] defines the bit pattern to be used for the start delimiter and end delimiter. The most significant byte defines the end delimiter and the least significant byte defines the start delimiter. For a good start and end delimiter this value shall be 0x8E71.

GATE_ERR	A single bit of the High Speed frame is transmitted without a Manchester Transition such that it is low throughout the bit time. hsLErrPosn and hsHErrPosn specify the high and low order words of the bit position where this occurs. Bit 0 is the first bit of the pre-amble. If GATE-HIGH is OR'd with hsHErrPosn then the bit will be transmitted high throughout the bit time.

If omitted, crMrtRtSa( ) uses NO_HS_ERRS as the default value.

NOTE: When defining errors you may define either a LS or HS error to be associated with the sub-address. If defining LS error then hsErrors must be set to NO_HS_ERRORS. If defining HS error then lsErrors must be set to NO_LS_ERRORS.

Example:

```
#include "drivers.h"

...

SaInfo       subAdrs;
DataInfo   data;
Sword       rtNum;
Uword      dataVals[96];
Error        error;
rtNum = 1 ;

/*
Setup RT1 HS TX SA 1  with a buffer of 96 words.   Tx the first 64 of these data words.
*/

data.size = 96 ;
data.action = NEW ;
data.data = dataVals;

for (i=0;i<data.size;i++)
   data.data[i] = i ;

subAdrs.sa = 1 ;
subAdrs.wrap = NO_WRAP ;
subAdrs.rxWCnt = 0 ;
subAdrs.txWCnt = 64 ;
subAdrs.hsRiTm = 0 ;
subAdrs.hsTiTm = 0 ;
subAdrs.rxData = NULL;
subAdrs.txData = &data;
subAdrs.rxErrData = NULL;
subAdrs.txErrData = NULL;

error = crMrtRtSa (&card1, rtNum, &subAdrs, HS_SA) ;
```

6.5. modMrtRtSa( )

Error modMrtRtSa(MemMapping *cardHandle, Sword rtNum,
SaInfo *sa, Sword saType)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
rtNum	The number of the RT to be modified.
sa	Pointer to a structure of type SaInfo which contains information related to the sub-address / mode code.
saType	The type of sub-address. Permitted values are: LS_SA Low speed sub-address HS_SA High speed sub-address

Description

The modMrtRtSa ( ) function modifies the unique data buffers and error injection which have been previously assigned to an RT/SA pair on the card identified by cardHandle. It is essential that cardHandle has been initialised via a previous call to setup( ) and that this RT/SA pair has already been setup.

sa is a pointer to a SaInfo structure. If sa is NULL then this is an error. The SaInfo type is declared as:

```
typedef struct [  Sword     sa;
                  Sword     wrap;
                  Sword     rxWCnt;
                  Sword     txWCnt;
                  Sword     hsRiTm;
                  Sword     hsTiTm;
                  DataInfo  *rxData;
                  DataInfo  *txData;
                  ErrorInfo  *rxErrData;
                  ErrorInfo  *txErrData;
               }  SaInfo;
```

sa	Sub-address number. sa contains the Low Speed sub-address, or High Speed sub-address number. This parameter is mandatory and must be set to a valid sub-address number.
wrap	Wrap type. wrap contains the wrap type. This specifies whether the RX and TX data buffers are common or not. It must be set to one of the following values:- DEF_WRAP Make no change to current wrapping WRAP RX and TX have the same data buffer NO_WRAP RX and TX have different data buffers

rxWCnt — Receive Word-count. For Low Speed transfers rxWCnt contains the number of words to be received. For High Speed transfers rxWCnt contains the number of words in the INFO field of the HS frame to be received. If rxWCnt equals DEFVAL then no change is made to the RX word-count. This parameter is mandatory.

txWCnt — Transmit Word-count. For Low Speed transfers txWCnt contains the number of words to be transmitted. For High Speed transfers txWCnt contains the number of words in the INFO field of the HS frame to be transmitted. If txWCnt equals DEFVAL then no change is made to the TX word-count. This parameter is mandatory.

hsRiTm — HS Receiver Initialisation Timeout. hsRiTm contains the High Speed Receiver Initialisation Time in uS. If hsRiTm equals DEFVAL then no change is made to the receiver Initialisation Time. This parameter is mandatory.

hsTiTm — HS Transmitter Initialise Time. hsTiTm contains the High Speed Transmitter Initialise Time in uS. If hsTiTm equals DEFVAL then no change is made to the Transmitter Initialise Time. This parameter is mandatory.

rxData — RX Data. rxData contains a pointer to a DataInfo structure which defines the data associated with this RX sub-address. Should rxData be NULL then no change is made to this data. If not NULL then a new data buffer is allocated for this sub-address and the size is sufficient to hold the number of words specified by rxWCnt. The elements of rxData must be setup if rxData is not NULL.

The DataInfo type is declared as :-

```
typedef struct   { Sword   size ;
                   Sword   posn ;
                   Sword   action ;
                   Uword   *data ;
                   } DataInfo ;
```

size specifies the size in words to be modified.

posn is the offset in the data buffer where modification begins. A value of zero is the first data word. For high speed message data buffers the value -3 indicates the FCPA word, -2 the DA and -1 the WC.

action specifies the action on the data buffer - this must be set to one of the following:

NEW	A new data buffer is created and filled with the data from the array pointed to by 'data'. In this case the value of 'posn' is ignored. 'size' words are copied.
OVERWRITE	Existing data starting at 'posn' and continuing for 'size' is overwritten by the data in the array pointed to by 'data' The value at data[0] is copied to position 'posn' in the data array.
APPEND	Data in the array pointed to by 'data' is appended to the data buffer. In this case the value of 'posn' is ignored. 'size' words are copied.
INSERT	Data in the array pointed to by 'data' of length 'size' words is inserted into the data buffer at position 'posn', existing data being moved for the insertion.

'data' is a pointer to an array of words which are used to modify the existing data buffer.

txData

TX Data. txData contains a pointer to a DataInfo structure which defines the data associated with this TX sub-address. Should txData be NULL then no change is made to this data. If not NULL then a new data buffer is allocated for this sub-address and the size is sufficient to hold the number of words specified by txWCnt. The elements of txData must be setup if txData is not NULL.

The DataInfo type is declared as :-

```
typedef struct   { Sword   size ;
                   Sword   posn ;
                   Sword   action ;
                   Uword   *data ;
                   } DataInfo ;
```

size specifies the size in words to be modified.

posn is the offset in the data buffer where modification begins. A value of zero is the first data word. For high speed message data buffers the value -3 indicates the FCPA word, -2 the DA and -1 the WC. This parameter relates only to the OVERWRITE and INSERT actions.

action specifies the action on the data buffer - this must be set to one of the following:

NEW	A new data buffer is created and filled with the data from the array pointed to by 'data'. In this case the value of 'posn' is ignored. 'size' words are copied.
OVERWRITE	Existing data starting at 'posn' and continuing for 'size' is overwritten by the data in the array pointed to by 'data' The value at data[0] is copied to position 'posn' in the data array.
APPEND	Data in the array pointed to by 'data' is appended to the data buffer. In this case the value of 'posn' is ignored. 'size' words are copied.
INSERT	Data in the array pointed to by 'data' of length 'size' words is inserted into the data buffer at position 'posn', existing data being moved for the insertion.

'data' is a pointer to an array of words which are used to modify the existing data buffer.

rxErrData — RX Error Data. rxErrData contains a pointer to an ErrInfo structure which defines the error associated with this RX sub-address. Should rxErrData be NULL then no change is made to the RX error injection.

The ErrInfo type is declared as :-

```
typedef struct   { Uword   lsErrors;
                   Uword   lsErrInfo;
                   Uword   lsErrPosn;
                   Uword   hsErrors;
                   Uword   hsErrInfo;
                   Uword   hsHErrPosn;
                   Uword   hsLErrPosn;
                 } ErrInfo;
```

lsErrors contains the type of error to be injected in the Low Speed transfer. This must be one of the following:-

NO_CHANGE	No change to LS error injection.
LS_ERRS	Low Speed error injection disabled
PARITY_ERR	Inject a parity error
MANCHESTER_ERR	Inject a Manchester error
SYNCHRO_ERR	Inject a synchro. error
WRD_LEN_ERR	Inject a word length error
WRONG_BUS_ERR	Inject a wrong bus error
BOTH_BUS_ERR	Transmit on both buses
RESP_TM_ERR	Inject a response time error

lsErrInfo contains extra Low Speed error information which depends on the value of lsErrors.

lsErrors	lsErrInfo
MANCHESTER_ERR	Bit Position in the word where error is injected. Zero is the most significant bit.
SYNCHRO_ERR	The desired sync pattern. This is 6 bits each representing 0.5 uS of the sync period. A '1' forces the sync high, '0' forces it low. A pattern of 0 is not allowed.
WORD_LEN_ERR	The desired word length in bits. A value of 16 is the "normal" word length, 15 is a word short by 1 bit.
RESP_TM_ERR	The required RT response time in microseconds.
POS_WRD_CNT_ERR	The number of extra words transmitted in the message.
NEG_WRD_CNT_ERR	The number of words that will be omitted from the message.

lsErrPosn contains the word number where the error will be injected when lsErrors is PARITY_ERR, SYNCHRO_ERR or WRD_LEN_ERR. For an RX message an error can only be injected into the status word. Thus lsErrPosn must be set to zero.

hsErrors contains the type of error to be injected in the High Speed transfer. If this is set to NO_CHANGE then no change will be made to the current HS error injection. If it is set to NO_HS_ERRS then HS error injection is disabled otherwise one or a combination of HS errors may be injected:

PRE_BIT_CNT_ERR	The number of pre-amble bits which is normally set to 40 is set to the value contained in hsErrInfo
NEG_3910_WRD_CNT_ERR	The High Speed Frame is transmitted with one word less than normal.
POS_3910_WRD_CNT_ERR	The High Speed Frame is transmitted with one extra word appended.
FCS_ERR	The Frame Check Sequence is transmitted with an invalid value.

HS_BIT_CNT_ERR

The number of bits to be removed from the HS data stream to create a bit count error. The number of bits to be removed is a value of 0-15 as defined in info[HsErrInfo]

HS_SD_ED_ERR

The contents of info[HsErrInfo] defines the bit pattern to be used for the start delimiter and end delimiter. The most significant byte defines the end delimiter and the least significant byte defines the start delimiter. For a good start and end delimiter this value shall be 0x8E71.

GATE_ERR	A single bit of the High Speed frame is transmitted without a Manchester Transition such that it is low throughout the bit time. hsLErrPosn and hsHErrPosn specify the high and low order words of the bit position where this occurs. Bit 0 is the first bit of the pre-amble. If GATE-HIGH is OR'd with hsHErrPosn then the bit will be transmitted high throughout the bit time.

NOTE: When defining errors you may define either a LS or HS error to be associated with the sub-address. If defining LS error then hsErrors must be set to NO_HS_ERRORS. If defining HS error then lsErrors must be set to NO_LS_ERRORS.

txErrData TX Error Data. txErrData contains a pointer to an ErrInfo structure which defines the error associated with this TX sub-address. Should txErrData be NULL then no change is made to the TX error injection.

The ErrInfo type is declared as :-

```
typedef struct {  Uword   lsErrors;
                  Uword   lsErrInfo;
                  Uword   lsErrPosn;
                  Uword   hsErrors;
                  Uword   hsErrInfo;
                  Uword   hsHErrPosn;
                  Uword   hsLErrPosn;
                  }  ErrInfo;
```

lsErrors contains the type of Low Speed error to be injected in the transfer. This must be one of the following:-

NO_CHANGE	No change to LS error injection.
LS_ERRS	Low Speed error injection disabled
PARITY_ERR	Inject a parity error
MANCHESTER_ERR	Inject a Manchester error
SYNCHRO_ERR	Inject a synchro. error
WRD_LEN_ERR	Inject a word length error
WRONG_BUS_ERR	Inject a wrong bus error
BOTH_BUS_ERR	Transmit on both buses
RESP_TM_ERR	Inject a response time error
POS_WRD_CNT_ERR	Inject a positive word count error
NEG_WRD_CNT_ERR	Inject a negative word count error

lsErrInfo contains extra Low Speed error information which depends on the value of lsErrors.

lsErrors	lsErrInfo
MANCHESTER_ERR	Bit Position in the word where error is injected. 0 is the most significant bit.
SYNCHRO_ERR	The desired sync pattern. This is 6 bits each representing 0.5 uS of the sync period. A '1' forces the sync high, '0' forces it low. A pattern of 0 is not allowed.
WORD_LEN_ERR	The desired word length in bits. A value of 16 is the "normal" word length, 15 is a word short by 1 bit.
RESP_TM_ERR	The required RT response time in microseconds.
POS_WRD_CNT_ERR	The number of extra words transmitted in the message.
NEG_WRD_CNT_ERR	The number of words that will be omitted from the message.

lsErrPosn contains the word number where the error will be injected when lsErrors is PARITY_ERR, SYNCHRO_ERR or WRD_LEN_ERR. Word zero is the status word, word 1 the first data word, etc.

hsErrors contains the type of error to be injected in the High Speed transfer. If this is set to NO_CAHNGE then no change is made to current HS error injection. If is set to NO_HS_ERRS then HS error injection is disabled otherwise one or a combination of the HS errors may be injected, as follows:-

PRE_BIT_CNT_ERR	The number of pre-amble bits which is normally set to 40 is set to the value contained in hsErrInfo
NEG_3910_WRD_CNT_ERR	The High Speed Frame is transmitted with one word less than normal.
POS_3910_WRD_CNT_ERR	The High Speed Frame is transmitted with one extra word appended.
FCS_ERR	The Frame Check Sequence is transmitted with an invalid value.
GATE_ERR	A single bit of the High Speed frame is transmitted without a Manchester Transition such that it is low throughout the bit time. hsLErrPosn and hsHErrPosn specify the high and low order words of the bit position where this occurs. Bit 0 is the first bit of the pre-amble. If GATE-HIGH is OR'ed with hsHErrPosn then the bit will be transmitted high throughout the bit time.

NOTE: When defining errors you may define either a LS or HS error to be associated with the sub-address. If defining LS error then hsErrors must be set to NO_HS_ERRORS. If defining HS error then lsErrors must be set to NO_LS_ERRORS.

Example:

```
#include "drivers.h"
...
SaInfo      subAdrs;
DataInfo    data;
Sword       rtNum;
Uword       dataVals[10];
Error       error;
rtNum = 1 ;
/*
Assuming that RT1 HS TX SA 1 has been setup with a buffer of 96 words ( but transmitting     only 64)
modify it to transmit all 96.  Also OVERWRITE the first 2 data words with data 1234 hex and 5678 hex..
*/
data.size = 2 ;
data.posn = 0
data.action = OVERWRITE
data.data = dataVals;
data.data[0] = 0x1234;
data.data[1] = 0x5678;

subAdrs.sa = 1;
subAdrs.wrap = DEF_WRAP ;
subAdrs.rxWCnt = DEFVAL ;
subAdrs.txWCnt = 96 ;
subAdrs.hsRiTm = DEFVAL;
subAdrs.hsTiTm = DEFVAL ;
subAdrs.rxData = NULL;
subAdrs.txData = &data;
subAdrs.rxErrData = NULL;
subAdrs.txErrData = NULL;
error = modMrtRtSa (&card1, rtNum, &subAdrs, HS_SA) ;
```

6.6. rdMrtRtSa( )

Error rdMrtRtSa(MemMapping *cardHandle, Sword rtNum,
SaInfo *sa, Sword saType)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
rtNum	The number of the RT to be read.
sa	Pointer to a structure of type SaInfo which contains information related to the sub-address / mode code.
saType	The type of sub-address. Permitted values are: LS_SA Low speed sub-address HS_SA High speed sub-address

Description:

The rdMrtRtSa ( ) function reads information related to this high speed or low speed RT/SA pair from the card identified by cardHandle. It is essential that cardHandle has been initialised via a previous call to setup( ) and that this RT/SA pair has already been setup.

sa is a pointer to a SaInfo structure. If sa is NULL then this is an error. The SaInfo type is declared as:

```
typedef struct {  Sword     sa;
                  Sword     wrap;
                  Sword     rxWCnt;
                  Sword     txWCnt;
                  Sword     hsRiTm;
                  Sword     hsTiTm;
                  DataInfo          *rxData;
                  DataInfo          *txData;
                  ErrorInfo  *rxErrData;
                  ErrorInfo  *txErrData;
               }  SaInfo;
```

sa — Sub-address number. sa contains the Low Speed sub-address, or High Speed sub-address number. This parameter is MANDATORY and must be set to a valid sub-address number.

wrap — Wrap type. wrap will be filled in by rdMrtRtSa ( ), and will contain one of the following values:-

WRAP	RX and TX have the same data buffer
NO_WRAP	RX and TX have different data buffers

rxWCnt — Receive Word-count. This will be filled in by rdMrtRtSa ( ), and for r Low Speed transfers rxWCnt will contain the number of words to be received. For High Speed transfers rxWCnt will contain the number of words in the INFO field of the HS frame to be received.

txWCnt — Transmit Word-count. This will be filled in by rdMrtRtSa ( ), and for Low Speed transfers txWCnt will contain the number of words to be transmitted. For High Speed transfers txWCnt will contain the number of words in the INFO field of the HS frame to be transmitted.

hsRiTm — HS Receiver Initialisation Timeout. hsRiTm will be filled in by rdMrtRtSa ( ), and will contain the High Speed Receiver Initialisation Time in uS. For low speed transfers hsRiTm will contain the value zero.

hsTiTm — HS Transmitter Initialise Time. hsTiTm will be filled in by rdMrtRtSa ( ), and will contain the High Speed Transmitter Initialise Time in uS. For low speed transfers hsTiTm will contain the value zero.

rxData — RX Data. rxData contains a pointer to a DataInfo structure, some of which will be filled in by rdMrtRtSa ( ). Should rxData be NULL then no RX data will be read.

The DataInfo type is declared as :-

```
typedef struct   { Sword   size ;
                   Sword   posn ;
                   Sword   action ;
                   Uword   *data ;
                 } DataInfo ;
```

size is set by the user to specify the maximum number of words which will be written by rdMrtRtSa ( ) into the data( ) array. rdMrtRtSa ( ) will set 'size' to the number of words it has copied into the data array.

posn is set by the user to indicate the start position in the data buffer from where the data must be read. A value of zero is the first data word. For high speed message data buffers the value -3 indicates the FCPA word, -2 the DA and -1 the WC.

action is unused.

'data' is a pointer to an array of words into which rdMrtRtSa ( ) copies the data associated with the message. The first word in the ‘data’ array will be the value of the associated DDB Data Status Report (DDB+6). The actual data will start at data[1].

txData — TX Data. txData contains a pointer to a DataInfo structure, some of which will be filled in by rdMrtRtSa ( ). Should txData be NULL then no TX data will be

read.

The DataInfo type is declared as :-

```
typedef struct   { Sword   size ;
                   Sword   posn ;
                   Sword   action ;
                  ..Uword  *data ;
                  } DataInfo ;
```

size is set by the user to specify the maximum number of words which will be written by rdMrtRtSa ( ) into the data( ) array. rdMrtRtSa ( ) will set 'size' to the number of words it has copied into the data array.

posn is set by the user to indicate the start position in data buffer from where the data must be read. A value of zero is the first data word. For high speed message data buffers the value -3 indicates the FCPA word, -2 the DA and -1 the WC.

action is unused.

'data' is a pointer to an array of words into which rdMrtRtSa ( ) copies the data associated with the message. The first word in the 'data' array will be the value of the associated DDB Data Status Report (DDB+6). The actual data will start at data[1].

rxErrData

RX Error Data. rxErrData contains a pointer to an ErrInfo structure. Should rxErrData be NULL then no RX error information will be read.

The ErrInfo type is declared as :-

```
typedef struct   { Uword   lsErrors;
                  ..Uword   lsErrInfo;
                  ..Uword   lsErrPosn;
                  ..Uword   hsErrors;
                  ..Uword   hsErrInfo;
                  ..Uword   hsHErrPosn;
                  ..Uword   hsLErrPosn;
                  } ErrInfo;
```

lsErrors will be filled in by rdMrtRtSa ( ) and will contain one of the following:-

NO_LS_ERRS	Low Speed error injection disabled
PARITY_ERR	Inject a parity error
MANCHESTER_ERR	Inject a Manchester error
SYNCHRO_ERR	Inject a synchro. error
WRD_LEN_ERR	Inject a word length error
WRONG_BUS_ERR	Inject a wrong bus error
BOTH_BUS_ERR	Transmit on both buses
RESP_TM_ERR	Inject a response time error
POS_WRD_CNT_ERR	Inject a positive word-count error
NEG_WRD_CNT_ERR	Inject a negative word-count error

lsErrInfo will be filled in by rdMrtRtSa ( ) and will contain extra Low Speed error information which depends on the value of lsErrors.

lsErrors	lsErrInfo
MANCHESTER_ERR	Bit Position in the word where error is injected. Zero is the most significant bit.

SYNCHRO_ERR	The desired sync pattern. This is 6 bits each representing 0.5 uS of the sync period. A '1' forces the sync high, '0' forces it low. A pattern of 0 is not allowed.
WORD_LEN_ERR	The desired word length in bits. A value of 16 is the "normal" word length, 15 is a word short by 1 bit.
RESP_TM_ERR	The required RT response time in microseconds.
POS_WRD_CNT_ERR	The number of extra words transmitted in the message.
NEG_WRD_CNT_ERR	The number of words that will be omitted from the message.

lsErrPosn will be filled in by rdMrtRtSa ( ) and will contain the word number where the error was injected when lsErrors is PARITY_ERR, SYNCHRO_ERR or WRD_LEN_ERR. For an RX message an error can only be injected into the status word. Thus if lsErrPosn is not zero the error is not relevant.

hsErrors will be filled in by rdMrtRtSa ( ) and will contain the type of error being injected in the High Speed transfer. This will be NO_HS_ERRS or a combination of the following:

PRE_BIT_CNT_ERR	The number of pre-amble bits will be contained in hsErrInfo
NEG_3910_WRD_CNT_ERR	The High Speed Frame is being transmitted with one word less than normal.
POS_3910_WRD_CNT_ERR	The High Speed Frame is being transmitted with one extra word appended.
FCS_ERR	The Frame Check Sequence is being transmitted with an invalid value.
HS_BIT_CNT_ERR	The frame is transmitted with a bit count error. The number of bits removed from the data will be contained in hsErrInfo
HS_SD_ED_ERR	The frame is transmitted with an invalid START and/or END delimiter. The delimiter patterns will be contained in hsErrInfo.
GATE_ERR	A single bit of the High Speed frame is being transmitted without a Manchester Transition such that it is low throughout the bit time. hsLErrPosn and hsHErrPosn will contain the high and low order words of the bit position where this occurs. Bit zero is the first bit of the pre-amble. If GATE-HIGH is OR'd with hsHErrPosn then the bit is being transmitted high throughout the bit time.

txErrData TX Error Data. txErrData contains a pointer to an ErrInfo structure. Should txErrData be NULL then no TX error information will be read.

The ErrInfo type is declared as :-

```
typedef struct {  Uword   lsErrors;
                  Uword   lsErrInfo;
                  Uword   lsErrPosn;
                  Uword   hsErrors;
                  Uword   hsErrInfo;
                  Uword   hsHErrPosn;
                  Uword   hsLErrPosn;
               }  ErrInfo;
```

lsErrors will be filled in by rdMrtRtSa ( ) and will contain one of the following:-

NO_LS_ERRS	Low Speed error injection disabled
PARITY_ERR	Inject a parity error
MANCHESTER_ERR	Inject a Manchester error
SYNCHRO_ERR	Inject a synchro. error
WRD_LEN_ERR	Inject a word length error
WRONG_BUS_ERR	Inject a wrong bus error
BOTH_BUS_ERR	Transmit on both buses
RESP_TM_ERR	Inject a response time error
POS_WRD_CNT_ERR	Inject a +ve word count error
NEG_WRD_CNT_ERR	Inject a -ve word count error

lsErrInfo will be filled in by rdMrtRtSa ( ) and will contain extra Low Speed error information which depends on the value of txErrData.lsErrors.

lsErrors	lsErrInfo
MANCHESTER_ERR	Bit Position in the word where error is injected. 0 is the most significant bit.
SYNCHRO_ERR	The desired sync pattern. This is 6 bits each representing 0.5 uS of the sync period. A '1' forces the sync high, '0' forces it low. A pattern of 0 is not allowed.
WORD_LEN_ERR	The desired word length in bits. A value of 16 is the "normal" word length, 15 is a word short by 1 bit.
RESP_TM_ERR	The required RT response time in microseconds.
POS_WRD_CNT_ERR	The number of extra words transmitted in the message.
NEG_WRD_CNT_ERR	The number of words that will be omitted from the message.

lsErrPosn will be filled in by rdMrtRtSa ( ) and will contain the word number where the error was injected when lsErrors is PARITY_ERR, SYNCHRO_ERR or WRD_LEN_ERR. Word zero is the status word, word 1 the first data word, etc.

hsErrors will be filled in by rdMrtRtSa ( ) and will contain the type of error being injected in the High Speed transfer. This will be NO_HS_ERRS or a combination of the following:-

PRE_BIT_CNT_ERR	The number of pre-amble bits will be in hsErrInfo
NEG_3910_WRD_CNT_ERR	The High Speed Frame is being transmitted with one word less than normal.
POS_3910_WRD_CNT_ERR	The High Speed Frame is being transmitted with one extra word appended.
FCS_ERR	The Frame Check Sequence is being transmitted with an invalid value.
HS_BIT_CNT_ERR	The frame is transmitted with a bit count error. The number of bits removed from the data will be contained in hsErrInfo
HS_SD_ED_ERR	The frame is transmitted with an invalid START and/or END delimiter. The delimiter patterns will be contained in hsErrInfo.
GATE_ERR	A single bit of the High Speed frame is being transmitted without a Manchester Transition such that it is low throughout the bit time. hsLErrPosn and hsHErrPosn will contain the high and low order words of the bit position where this occurs. Bit zero is the first bit of the pre-amble. If GATE-HIGH is ORed with hsHErrPosn then the bit is being transmitted high throughout the bit time.

Example:

```
#include "drivers.h"

...

SaInfo       subAdrs;
DataInfo   data;
Sword       rtNum;
Uword      dataVals[64];
Error       error;
rtNum = 1 ;

 /*
  Read HS data blocks 2 and 3 from RT1 HS TX SA 1
 */
 data.size = 64 ;
 data.posn = 32;
 data.data = dataVals;

 subAdrs.sa = 1;
 subAdrs.rxData = NULL;
 subAdrs.txData = &data;
 subAdrs.rxErrData = NULL;
 subAdrs.txErrData = NULL ;
 error = rdMrtRtSa (&card1, rtNum, &subAdrs, HS_SA) ;
```

6.7. modMrtRtMd( )

Error modMrtRtMd(MemMapping *cardHandle, Sword rtNum, MdInfo *md)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
rtNum	The number of the RT to be modified.
md	Pointer to a structure of type MdInfo which contains information related to this mode code.

Description:

The modMrtRtMd( ) function modifies the error injection related to an RT/MODE CODE pair on the card identified by cardHandle. It is essential that cardHandle has been initialised via a previous call to setup( ). rtNum is a signed 16-bit quantity. It is set by the user and must contain a valid RT number. This parameter is MANDATORY.

md is a pointer to a MdInfo structure. If md is NULL then this is an error.

The MdInfo type is declared as

```
typedef struct { Sword       md;
                 MdErrInfo   *errData;
                 Uword       *dBuff[6];
               } MdInfo;
```

md — Mode code number. md is set by the user and contains the mode code number. This element is mandatory and must be set to a valid mode code number.

dBuff — Not used.

errData — Error Data. errData contains a pointer to an MdErrInfo structure which defines the error associated with the mode code. Should errData be NULL then no change is made to the error injection.

The MdErrInfo type is declared as :-

```
typedef struct {  Sword  lsErrors ;
                  Sword  lsErrInfo ;
                  Sword  lsErrPosn ;
               } MdErrInfo ;
```

lsErrors contains the type of error to be injected into the transfer. This must be one of the following:-

NO_CHANGE	Make no change to LS error injection
NO_LS_ERRS	Low Speed error injection disabled
PARITY_ERR	Inject a parity error
MANCHESTER_ERR	Inject a Manchester error
SYNCHRO_ERR	Inject a synchro. error
WRD_LEN_ERR	Inject a word length error
WRONG_BUS_ERR	Inject a wrong bus error
BOTH_BUS_ERR	Transmit on both buses
RESP_TM_ERR	Inject a response time error

lsErrInfo contains extra Low Speed error information which depends on the value of lsErrors.

lsErrors	lsErrInfo
MANCHESTER_ERR	Bit Position in the word where error is injected. Zero is the most significant bit.
SYNCHRO_ERR	The desired sync pattern. This is 6 bits each representing 0.5 uS of the sync period. A '1' forces the sync high, '0' forces it low. A pattern of 0 is not allowed.
WORD_LEN_ERR	The desired word length in bits. A value of 16 is the "normal" word length, 15 is a word short by 1 bit.

lsErrPosn contains the word number where the error will be injected when lsErrors is PARITY_ERR, SYNCHRO_ERR or WRD_LEN_ERR. For an RX mode code an error can only be injected into the status word. Thus lsErrPosn must be set to 0. For a TX mode code with associated data word an error can be injected into the data word by setting lsErrPosn to 1.

Example:

```
#include "drivers.h"
      ...
MdErrInfo    errData;
Sword        rtNum
Error        error;

/*
Inject a parity error into the Vector Word transmitted
when Transmit Vector Word Mode Code is sent to RT 1
*/
 rtNum               = 1;
mode.md              = 16;
mode.errData         = &errData;
errData.lsErrors     = PARITY_ERR ;
errData.lsErrPosn    = 1;
error = modMrtRtMd (&card1, rtNum, &mode) ;
```

6.8. rdMrtRtMd( )

Error rdMrtRtMd(MemMapping *cardHandle, Sword rtNum, MdInfo *md)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
rtNum	The number of the RT to be read.
md	Pointer to a structure of type MdInfo.

Description

The rdMrtRtMd( ) function reads information related to this RT/MODE CODE pair from the card identified by cardHandle. It is essential that cardHandle has been initialised via a previous call to setup( ). rtNum is a signed 16-bit quantity. It is set by the user and must contain a valid RT number. This parameter is mandatory.

md is a pointer to a MdInfo structure. If md is NULL then no information is read from the card.

The MdInfo type is declared as

```
typedef struct { Sword       md;
                 MdErrInfo   *errData;
                 Uword       dBuff[6];
                 } MdInfo;
```

md	Mode code number. md is set by the user and contains the mode code number. This element is mandatory and must be set to a valid mode code number.
dBuff	After execution of this function dBuff[0] will contain the value of the status report in the associated DDB (offset +6). The elements dBuff[1] to dBuff[5] will contain any RX data that has been received due to this mode code. Under normal circumstances, only dBuff[1] will be used.
errData	Error Data. errData contains a pointer to an MdErrInfo structure. Should errData be NULL then no error information will be read.

The MdErrInfo type is declared as :-

```
typedef struct { Sword   lsErrors ;
                 Sword   lsErrInfo ;
                 Sword   lsErrPosn ;
                 } MdErrInfo ;
```

lsErrors will be filled in by rdMrtRtMd( ) and will contain one of the following:-

NO_CHANGE	Make no change to LS error injection
NO_LS_ERRS	Low Speed error injection disabled
PARITY_ERR	Inject a parity error
MANCHESTER_ERR	Inject a Manchester error
SYNCHRO_ERR	Inject a synchro. error
WRD_LEN_ERR	Inject a word length error
WRONG_BUS_ERR	Inject a wrong bus error
BOTH_BUS_ERR	Transmit on both buses
RESP_TM_ERR	Inject a response time error

lsErrInfo will be filled in by rdMrtRtMd( ) and will contain extra Low Speed error information which depends on the value of errData.lsErrors.

lsErrors	lsErrInfo
MANCHESTER_ERR	Bit Position in the word where error is injected. Zero is the most significant bit.
SYNCHRO_ERR	The desired sync pattern. This is 6 bits each representing 0.5 uS of the sync period. A '1' forces the sync high, '0' forces it low. A pattern of 0 is not allowed.
WORD_LEN_ERR	The desired word length in bits. A value of 16 is the "normal" word length, 15 is a word short by 1 bit.

lsErrPosn will be filled in by rdMrtRtMd( ) and will contain the word number where the error was injected when lsErrors is PARITY_ERR, SYNCHRO_ERR or WRD_LEN_ERR. For an RX mode code an error can only be injected into the status word. Thus if lsErrPosn is not zero the error is not relevant. For a TX mode code with associated data word an error can be injected into the data word thus if lsErrPosn is neither zero or to 1 the error is not relevant.

Example:

```
#include "drivers.h"
...
MdErrInfo    errData;
Sword        rtNum;
Error        error;
/*
 Read the error information associated with RT 1
 Transmit Vector Word Mode Code.
*/
rtNum            = 1 ;
mode.md          = 16;
mode.errData     = &errData;
error = rdMrtRtMd (&card1, rtNum, &mode) ;
```

6.9. srqQueue( )

Error srqQueue(MemMapping *cardHandle, SrqInfo *srqData)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
*srqData	Pointer to a structure of type SrqInfo describing the SRQ queue insertions required.

The elements of the SrqInfo structure is as follows:

RtNo	RT number
vector	Vector word to be transmitted for TX vector mode code
action	SRQ_Q_CLEAR - Set all values in the SRQ Queue to 0x0000 SRQ_Q_ENTRY - Put an entry into the SRQ Queue
priority	Q_HI_PRIORITY or Q_LO_PRIORITY
Result	Returned none-zero if the function did not make an entry in the queue. This should only happen if the action is SRQ_Q_CLEAR or the action is SRQ_Q_ENTRY and the queue is full

Description:

The SRQ_Q_CLEAR option is to enable the user to clear the queue prior to run-time. When an entry is placed on the queue the RT will respond with the SRQ bit set. When the BC polls the RT for a TX vector word the RT will respond with the vector word as defined in the SrqInfo structure.

Example:

```
#include "drivers.h"
...
SrqInfo srqData;
Error        error;
…
srqData.rtNo = 2;
srqData.vector = 0x0010;
srqData.action = SRQ_Q_ENTRY;
srqData.priority = Q_HI_PRIORITY;
error = srqQueue(&card1, &srqData);
```

6.10. joinMrt( )

Error joinMrt(MemMapping *cardHandle, JoinInfo *join)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
*join	Pointer to structure of type JoinInfo describing which RT sub-addresses to join.

The elements of the JoinInfo structure are as follows:

rtNum1	1st RT address
sa1	1st RT subaddress
txrx1	TX_TYPE or RX_TYPE buffer
rtNum2	2nd RT Address
sa2	2nd RT subaddress
txrx2	TX_TYPE or RX_TYPE buffer
buffType	Buffer type - LS_SA or HS_SA
copyErr	Set if copy of errors to 2nd RT subaddress is to be made

Description:

This function takes the DDB pointer in the lookup table for rtNum1, sa1 and saves it in the lookup table DDB pointer location for rtNum2, sa2. The type of buffer LS or HS is defined by buffType. The TX or RX lookup table positions are defined by txrx1 and txrx2. If copyErr is set TRUE then the error word will also be copied to the rtNum2, sa2 lookup table.

Note:

- This function allows any previous definition (single buffer, linked buffer or extended subaddress) to be shared by another RT subaddress. In the case of the extended subaddress the copyErr must be set TRUE to copy the error word 0xC000 to the rtNum2, sa2 lookup table.

- WARNING -This function is outside the control of the drivers database. This function should NOT be called until all other set-ups have been carried out. All following read and writes to the shared buffers MUST be done via the rtNum1, sa1 parameters.

Example:

```
#include "drivers.h"
…
Error           error;
JoinInfo        join;
…
…
join.rtNum1     = 1;
join.rtNum2     = 2;
join.sa1        = 10;
join.sa2        = 11;
join.txrx1      = RX_TYPE;
join.txrx2      = RX_TYPE;
join.buffType   = LS_SA;
join.copyErr    = 0;

error  =  joinMsg ( &card1, &join);
```

6.11. linkMrt( )

Error linkMrt(MemMapping *cardHandle, Slong rtNo, Sword saMd,
Sword txrxType, LinkInfo *link)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
rtNo	The number of the RT
saMd	Subaddress or Mode Code number
txrxType	Buffer type - TXTYPE or RXTYPE
*link	Pointer to LinkInfo structure defining the link buffers.

The elements of the LinkInfo structure are as follows:

mode	Mode of operation for link. This shall be one of the following values: LINK_BUFF_ALWAYS The option mask link bits are set for always. LINK_BUFF_ON_GOOD The option mask link bits are set for good msg. LINK_BUFF_WRITE Write to buffers that have already been created. LINK_BUFF_READ Read buffers that have already been created. LINK_BUFF_INIT Initialise the buffer pointer to bCount
bCount	This shall define the total number of buffers required, or in the case of LINK_BUFF_INIT, the buffer number to be pointed at in the lookup table.
BuffType	This shall define the buffer type as:- LS_SA - Low speed 3838 type buffer LS_MD - Low speed 3838 mode code buffer HS_SA - High speed 3910 type buffer
wCount	This shall define the number of words to read from the buffers or write to the buffers.
**buffer	This is an array of 'bCount' pointers. Each pointer must be set to point to a buffer of 'wCount' words or set to NO_PTR.

Description

Creating buffers:

- First create an RT Subaddress or mode code using the standard driver functions, defining the data buffer size required.

- Now set mode for LINK_BUFF_ALWAYS or LINK_BUFF_ON_GOOD with the required number of buffers defined in 'bCount'.

- Now set 'buffers' to point to an array of 'bCount' pointers. Each pointer must point to a Uword array of data or set to the value NO_PTR. If the pointer is not NO_PTR then the data will be used to initialise the particular data buffer starting at the first data word. All buffer initialisation can be disabled by setting 'buffers' to NO_PTR.

- Set the buffer type, txrxType to TXTYPE or RXTYPE, the rtNo and saMd to the desired subaddress or mode code. Set wCount and call linkMrt(). A further bCount-1 buffers will now be created with each link pointing to the next DDB. The last DDB will point back to the 1st DDB that was created using the standard function. Each consecutive buffer, including the one that already existed, will be initialised with data values if the corresponding buffer[i] pointer is not set to NO_PTR.

Writing to buffers:

If further updating of buffers is required then this can be done by setting the mode to LINK_BUFF_WRITE and calling the function linkMsg(). The buffers pointed to by 'buffer' will be used to fill the data buffers on the card. If a buffer pointer is set to NO_PTR then the corresponding data buffer will not be written to.

Reading from buffers:

If a data buffer(s) is required to be read then set the corresponding buffer[i] pointer to a Uword array for storing the data. If you do not wish to read a particular buffer then set the pointer to NO_PTR. Now call linkMsg with mode set to LINK_BUFF_READ.

The read back includes header information as follows:

LS Buffer:	DDB Status	HS Buffer:	DDB Status
	TTAG High		TTAG High
	TTAG Low		TTAG Low
	Data 1		FC,PA
	|		DA
	Data n		WC
			Data 1
			|
			Data n

Note

- The crMrtRtSa() function creates the initial buffer. The linkMrt() MUST use the correct txrxType as was used for crMrtRtSa().

- The order of the buffers will be in the order in which they are linked. The 1st buffer, corresponding to buffer[0], will be the buffer that was initially created using the standard function.

- If the buffers are to be common to both TX and RX subaddress then the crMrtRtSa() function must create the initial buffer with the WRAP mode set.

- The LabVIEW version only allows the writing and reading of 1 buffer when in LINK_BUFF_READ and LINK_BUFF_WRITE only. A maximum of 20 buffers are allowed.

Initialising the lookup buffer pointer:

To initialise the lookup buffer, set the bCount value to the buffer number. The buffer numbers start at 1. Now set the mode to LINK_BUFF_INIT and execute the function.

Example:

```
#include "drivers.h"
…
Error      error;
LinkInfo   link;
Slong      id;
link.mode          = LINK_BUFF_ALWAYS;
link.bCount        = 8;
link.buffType      = LS_SA;
link.wCount        = 32;
link.buffer        = NO_PTR;

error  =  linkMrt( &card1, 1, 30,  RX_TYPE, &link);
```

6.12. extendMrt( )

Error extendMrt(MemMapping *cardHandle, Slong rtNum, Sword sa,
Sword txrxType, ExtInfo *extd)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
rtNum	The number of the RT
sa	RT subaddress number
txrxType	Buffer type - TXTYPE or RXTYPE
extd	Pointer to structure of type ExtInfo defining the extended subaddress parameters

The elements of the LinkInfo structure are as follows:

mode	Mode can be set to one of 3 values: EXTEND_SUB Create an extended subaddress lookup table EXTEND_SUB_WRITE Write to buffers that have already been created. EXTEND_SUB_READ Read buffers that have already been created
wrap	Can be set to one of 2 values: WRAP: Make TX and RX subaddress point to the extended subaddress lookup table NO_WRAP: Extend the txrxType only
wCount	This shall define the number of words to read from the buffers or write to the buffers.
**buffer	This is an array of 32 pointers. Each pointer must be set to point to a buffer of 'wCount' words or set to NO_PTR.

Description:

Creating extended subaddress:

- First create an RT Subaddress 3838 subaddress using crMrtRtSa(), defining the data buffer size required
- Now set mode for EXTEND_SUB
- Now set 'buffers' to point to an array of 32 pointers. Each pointer must point to a Uword array of data or set to the value NO_PTR. If the pointer is not NO_PTR then the data will be used to initialise the particular data buffer starting at the first data word. All buffer initialisation can be disabled by setting 'buffers' to NO_PTR
- Set the buffer type, txrxType to TXTYPE or RXTYPE, the rtNum and sa to the desired subaddress. Set wCount and call extendMrt(). A new lookup table will be created of 32 elements. Each element will have the initial error word as defined by crMrtRtSa() followed by a unique DDB address. The original error word in the LS lookup table will be changed to 0xC000 for extended subaddress mode. If the wrap is set to WRAP then the other LS lookup table entry TX or RX (depending on the value of txrxType) will also be modified to point to the extended lookup table.

Writing to buffers:

If further updating of buffers is required then this can be done by setting the mode to EXTEND_SUB_WRITE and calling the function extendMrt(). The buffers pointed to by 'buffer' will be used to fill the data buffers on the card. If a buffer pointer is set to NO_PTR then the corresponding data buffer will not be written to.

Reading from buffers:

If a data buffer(s) is required to be read then set the corresponding buffer[i] pointer to a Uword array for storing the data. If you do not wish to read a particular buffer then set the pointer to NO_PTR. Now call linkMsg with mode set to EXTEND_SUB_READ. The read back includes header information as follows:

LS Buffer:	DDB Status
	TTAG High
	TTAG Low
	Data 1
	|
	Data n

HS Buffer:	DDB Status
	TTAG High
	TTAG Low
	FC,PA
	DA
	WC
	Data 1
	|
	Data n

Note

- The crMrtRtSa() function creates the initial buffer. The extendMrt() MUST use the correct txrxType as was used for crMrtRtSa()

- The LabVIEW version only allows the writing and reading of 1 buffer when in EXTEND_SUB_READ and EXTEND_SUB_WRITE only. A maximum of 20 buffers are allowed.

Example:

```
#include "drivers.h"
…
Error      error;
ExtInfo    extd;
…
extd.mode       = EXTEND_SUB;
extd.wrap       = NO_WRAP;
extd.buffType   = LS_SA;
extd.wCount     = 32;
extd.buffer     = NO_PTR;
error           =  extendMrt( &card1, 1, 30, RX_TYPE,  &extd);
```

6.13. actMrt( )

Error actMrt(MemMapping *cardHandle)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).

Description

The actMrt( ) function activates the card when in MRT mode. It is essential that cardHandle has been initialised via a previous call to setup( ).

Example:

```
#include "drivers.h"
...
Error          error;
…
error = actMrt (&card1) ;
```

6.14. deActMrt( )

Error deActMrt(MemMapping *cardHandle)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).

Description

The deActMrt( ) function deactivates the card when in MRT mode. It is essential that cardHandle has been initialised via a previous call to setup( ).

Example:

```
#include "drivers.h"
...
Error          error;
…
error = deActMrt (&card1) ;
```

7. CHRONOLOGICAL MONITOR FUNCTIONS

7.1. INTRODUCTION

The Chronological Monitor functions manage the setup of the cards when in CM mode. Refer to Appendix E for an application example that uses the Chronological Monitor drivers.

The drivers setup a default trigger condition during setup( ). The default trigger is :

T1 XXX XXX hXXXX	T1 is set to trigger on any word
If T1 CONTINUE FINISH PTC 0	After any message trigger stop with no post trigger messages

7.2. defTrigs( )

Error defTrigs (MemMapping *cardHandle, Ubyte *trigger, Uword *result)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
trigger	Pointer to an ASCII character string defining the trigger(s) definition.
result	Pointer to a Uword. If the syntax of 'trigger' is incorrect then E_CM_TRIG_DEF_ERR is returned and this word shall contain the character position in the string where the error occurred.

Description:

The defTrigs( ) function sets up to 4 trigger definitions. The trigger definitions are defined by the character string 'trigger' as follows:-

T<n> <bus> <typ> <f><pattern> {[errors]}<z>

{ }	These indicate optional section in the line.
<n>	1 char Trigger number 1-4.
<bus>	3 chars PRI,SEC,BTH,XXX (BTH = Both buses, XXX = Don't care).
<typ>	3 chars CMD,STA,DAT,RTR,XXX (RTR = RT to RT, XXX = Don't care).
<f>	1 char, 'B' for binary pattern, 'H' for hex pattern, 'C' or 'S' for 1553 format.
<pattern>	16 chars for binary, 4 chars for hex. 1553 format - 1F 1 1F 1F. The T/R bit can be represented as T,R,1 or 0. For all options 'X' = don't care.

[errors] The errors are optional. If not used or no errors are placed between the '[ ]' the error condition will be don't care. The errors are as follows:-

Py	Parity error
Mn	Manchester error
Lg	Long word
Sh	Short word
Wc	Word count error
NR	No response
TA	Terminal address error
Sy	Sync error

If more than 1 of these errors is inserted between the brackets (e.g.[Py Mn]) the condition will be a ***logical or*** of the errors.

<z> Trigger pattern terminator. This can be any char if a new trigger pattern is to follow. If no more pattern descriptions are to follow this MUST be 0. The termination character must immediately follow the last field in the trigger definition.

NOTE:

- Up to 4 trigger patterns can be defined in the 1 string.
- Defining the same trigger twice in the same string will return an error.
- The parsing is NOT case sensitive.
- If the error E_CM_TRIG_DEF_ERR is returned, the value of (*result) shall point to the position of the error in the string 'trigger'.

Example:

```
#include "drivers.h"
...
Uword   result;
Error    error;
static    Ubyte trigger[ ] =
{
          "T1 Pri Cmd cXX T 01 XX [Mn Py] "
          "T2 Pri Dat hABCD "
          "T3 Sec Dat b0000XXXX11110000 [Lg Sh] "
          "T4 XXX Sta s01 1 00 00\0"
 };

error = defTrigs (&card1,trigger,&result);
```

7.3. defSeq( )

Error defSeq (MemMapping *cardHandle, Ubyte *sequence, Uword *result)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
sequence	Pointer to an ASCII character string defining the trigger sequence definition.
result	Pointer to a Uword. If the syntax of 'sequence' is incorrect such that the error E_CM_SEQ_DEF_ERR is returned this word shall contain the char position in the string where the error occurred.

Description:-

The defSeq( ) driver sets up the trigger sequence defined by the ASCII string 'sequence'. This string is a series of instruction lines defining, in a language type format, the required trigger sequence. This simple language has a set number of possible commands as follows:-

When hardware trigin <t> continue
If T<n> continue {else back <c>}
If not T<n> continue {else back <c>}
If T<n> and word <w> is T<n> continue
If T<n> and word <w> is not T<n> continue
If T<n> and word <w> is T<n> select <s> continue
If T<n> and word <w> is not T<n> select <s> continue
If T<n> and HS T<n> {errors[e]} continue
If T<n> and HS not T<n> {errors[e]} continue
Finish PTC <p> {[trigout on <x>]}

{ }	These indicate optional section to the command.
<n>	Trigger number 1-4.
<t>	'HI-LO' or 'LO-HI'.
<c>	Branch back <c> lines. This must be 0-3 and point to a valid line.
<w>	Word number in message for secondary trigger.
<s>	Number of messages to selectively capture.
<p>	Post trigger message count.
<x>	'COMPLETE' or 'TRIGGER'.
[e]	HS errors for trigger condition .

NOTE:

- The parsing of this string is NOT case sensitive.
- The { } define optional section. They are NOT in the string.
- The hardware trigin command, if used, MUST be the first line.
- The word number <w> can be 'X' defining don't care.
- The selective message count <s> can be replaced with 'FOREVER'. In this case the following 'continue' statement MUST NOT be used.
- The HS errors are optional. If not used or no errors are placed between the '[ ]' the error condition will be don't care.

  The errors are as follows:-

  'FCS' Frame check sequence error
  'HWC HS word count error
  'HNR' HS no response error
  'HBC' Bit count error in HS data
  'HBE' Bit error in HS data

  If more than 1 of these errors is inserted between the brackets (e.g. [FCS HWC]) the condition will be a logical or of the errors.

- The PTC <p> can be replaced with 'FOREVER' for continuous capture.
- If the error E_CM_TRIG_SEQ_ERR is returned, the value of (*result) shall point to the position of the error in the string 'sequence'.
- The termination character for each command line is '\n' with the exception of the 'Finish PTC' line which is terminated by '\0'.

Example:-

```
#include "drivers.h"
...
Uword    result;
Error    error;
static    Ubyte sequence[ ] =
{
        "When hardware trigin HI-LO continue\n"
        "If T1 and word 3 is T2 continue\n"
        "If T3 continue else back 1\n"
        "Finish PTC 100 [trigout on COMPLETE]\0"
};

error = defSeq (&card1,sequence,&result);
```

7.4. getRange( )

Error getRange (MemMapping *cardHandle, Slong *result)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
result	Pointer to an array of 3 Slong for storage of getRange( ) result.

Description:-

The getRange( ) function gets the range of captured messages available on the stack. The parameter 'result' is an array of 3 Slongs that is used by getRange( ) to report the range of the stack as follows:-

result[0] = Total number of stack messages.
result[1] = Maximum -ve message number.
result[2] = Maximum +ve message number.

NOTE: The trigger message is message zero.

Example:-

```
#include "drivers.h"
...
Slong   result[3];
Error    error;

error = getRange (&card1,result);
```

7.5. getMsg( )

Error getMsg (MemMapping *cardHandle, Slong msgNum, MsgInfo *msg)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
msgNum	Stack message number required.
msg	Pointer to a MsgInfo structure which defines where the message is to be stored and the size of the message buffer available.

Description:

The getMsg( ) function reads the message number 'msgNum' from the stack and stores it.

The MsgInfo type is declared as:-

```
typedef struct { Uword size;
Uword          *msgWords;
} MsgInfo;
```

size	specifies the maximum number of message words which will be written by getMsg( ) into the msgWords[ ] array. getMsg( ) will set this value to the number actually read which will be less than or equal to the value prior to calling the function. The values MAX_HS_MSG_SIZE and MAX_LS_MSG_SIZE are defined to facilitate the user when allocating memory for the messages.

msgWords points to a buffer which will be filled in by getMsg( ) with 'size' words.

The format of the message stored in msgWords[ ] is as follows:-

```
msg->msgWords[0]      = Previous address page pointer.

msg->msgWords[1]      = Time-Tag HI   (Non IRIG-B clock)
msg->msgWords[2]      = Time-Tag LO  (Non IRIG-B clock)

msg->msgWords[1]      = Time-Tag 1    (IRIG-B clock)
msg->msgWords[2]      = Time-Tag 2.   (IRIG-B clock)
msg->msgWords[3]      = Time-Tag 3.   (IRIG-B clock)
msg->msgWords[4]      = Time-Tag 4.   (IRIG-B clock)

msg->msgWords[x]      = Word 1 pattern  x = 5 for IRIG-B clock
msg->msgWords[x+1]    = Word 1 errors.   x = 3 for all others

msg->msgWords[n]      = Last Word pattern.
msg->msgWords[n+1]    = Last Word errors.
msg->msgWords[n+2]    = 1st RT response time.
msg->msgWords[n+3]    = 2nd RT response time.
msg->msgWords[n+4]    = Next address page pointer.
msg->msgWords[n+5]    = HS word-count register.
msg->msgWords[n+6]    = HS error word.
msg->msgWords[n+7]    = HS word-count error word.
msg->msgWords[n+8]    = HS word 1
```

NOTE:

- If the message does not contain HS data the value of the HS word-count register will be 0.
- If the message does have HS data the words will be stored in msg->msgWords[n+8] onwards.
- For detailed information of the various message elements see the user manual.

Example:

```
#include "drivers.h"
...
MsgInfo    msg;
Slong      msgNum;
Uword      buffer[MAX_HS_MSG_SIZE];
Error      error;

msgNum             = 10;
msg.size           = MAX_HS_MSG_SIZE;
msg.msgWords   = buffer;

error   = getMsg (&card1, msgNum, &msg);
```

7.6. findMsgs( )

Error findMsgs (MemMapping *cardHandle, Ubyte *pattern,
Uword *result, FindInfo *found)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
pattern	Pointer to an ASCII character string defining the word to be searched for.
result	Pointer to a Uword. If the syntax of 'pattern' is incorrect such that the error E_CM_SRCH_ERR is returned, this word shall contain the char position in the string where the error occurred.
found	Pointer to a FindInfo structure which defines where the message numbers are to be stored and the size of the buffer available.

Description:

The findMsgs( ) function searches through the stack and reports the message number of any message matching the ASCII string 'pattern'.

The FindInfo type is declared as:-

```
typedef   struct
          {
          Ulong   size;
          Slong   *msgNums;
          } FindInfo;
```

size	specifies the maximum number of message numbers which will be written by findMsgs( ) into the msgNums[ ] array. findMsgs( ) will set this value to the number of messages actually found and stored in msgNums[ ]. This value will be less than or equal to the value prior to calling the function.
msgNums	points to a buffer which will be filled in by findMsgs( ) with the message numbers found. These message numbers shall be in ascending order starting at the most -ve position of the stack.

The format of the 'pattern' string is as follows:-

<bus> <typ> <f><pattern> {[errors]}<z>

{ } These indicate optional section in the line.

<bus> 3 chars PRI,SEC,BTH,XXX
(BTH = Both buses, XXX = Don't care).

<typ> 3 chars CMD,STA,DAT,RTR,XXX (RTR = RT to RT, XXX = Don't care).

<f> 1 char, 'B' for binary pattern, 'H' for hex pattern, 'C' or 'S' for 1553 format.
<pattern> 16 chars for binary, 4 chars for hex. 1553 format - 1F 1 1F 1F.
The T/R bit can be represented as T, R, 1 or 0.
For all options 'X' = don't care.

[errors] The errors are optional. If not used or no errors are placed between the '[ ]' the error condition will be don't care.
The errors are as follows:-

Py	Parity error
Mn	Manchester error
Lg	Long word
Sh	Short word
Wc	Word count error
NR	No response
TA	Terminal address error
Sy	Sync error

If more than 1 of these errors is inserted between the brackets (e.g.[Py Mn]) the condition will be a logical or of the errors.

<z> Pattern terminator. This MUST be '\0'.

NOTE:

- The parsing is NOT case sensitive.
- If the error E_CM_SRCH_ERR is returned, the value of (*result) shall point to the position of the error in the string 'pattern'.

Example:-

```
 #include "drivers.h"
 ...
static       Ubyte pattern[ ] = {"Pri Dat hABCD\0" };
FindInfo    found;
Uword      result;
Slong      buffer[100];
Error       error;

found.size           = 100;
found.msgNums  = buffer;

error = findMsgs (&card1, pattern, &result, &found);
```

7.7. startCM( )

Error startCM (MemMapping *cardHandle)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).

Description

The startCM( ) starts the chronological monitor searching for the trigger condition.

Example

```
#include "drivers.h"
...
Error       error;
…
error = startCM(&card1);
```

7.8. stopCM( )

Error stopCM (MemMapping *cardHandle)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).

Description

The stopCM( ) stops the chronological monitor searching for the trigger condition or saving stack data.

Example

```
#include "drivers.h"
...
Error       error;
…
error = stopCM(&card1);
```

8. GENERAL FUNCTIONS

8.1. INTRODUCTION

The General Functions are common to the Bus Controller, Multi-Remote Terminal and Chronological monitor modes of operation of the card.

8.2. rdStatus( )

Error rdStatusReg (MemMapping *cardHandle, Uword *cardStatus)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
cardStatus	Pointer to location for Status Register value.

Description

The rdStatusReg ( ) function reads from the status register in the card identified by cardHandle. It is essential that cardHandle has been initialised via a previous call to setup( ).

Example

```
#include "drivers.h"
...
 Uword     cardStatus;
Error       error;
…
error = rdStatusReg (&card1,  &cardStatus);
```

8.3. exCmd( )

Error exCmd (MemMapping *cardHandle, Uword cmd)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
cmd	The command to be executed.

Description:

The exCmd( ) function executes a command on the card identified by cardHandle. It is essential that cardHandle has been initialised via a previous call to setup( ).

cmd is a 16-bit unsigned quantity which is the value to be written into the Command Register. The values can be entered as defined in the user manual. Alternatively the following constants are available from the drivers:-

GOTO_BC_MODE	Go to BC mode
GOTO_MRT_MODE	Go to MRT mode
GOTO_CM_MODE	Go to Chron Mon mode
BC_START	Start Bus Controller
MRT_START	Start MRT
CM_START	Start Chron Mon
BC_STOP	Stop Bus Controller
MRT_STOP	Stop MRT
CM_STOP	Stop Chron Mon
PAUSE	Pause the local clock
UNPAUSE	Unpause the local clock
SELFTEST	Execute selftest
LOAD_CLOCK	Load clock with value
SYNZ_CLOCK	Synchronise clock with value

Example:

```
#include "drivers.h"
…
Error   error;

error = exCmd (&card1,  CM_START);
```

8.4. setClk( )

Error setClk (MemMapping *cardHandle, Ulong clockValue, Uword mode)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
clockValue	Value to set in clock.
mode	Set to LOAD_CLOCK for load clock with value. Set to SYNZ_CLOCK for synchronise clock with value (add as offset)

Description:-

The setClk ( ) function takes the value of 'clockValue' and, if the mode is LOAD_CLOCK, loads the local clock with it. If the mode is SYNZ_CLOCK and the clock type is standard 32 bits the value is added to the current clock value as a signed offset.

For standard 32 bit clock the value of clockValue shall be a 32 bit binary number.

For IRIG-B type clock the format of 'clockValue' shall be as follows:

LL CC DDDDDDDD HHHHHH MMMMMMM SSSSSSS

LL	Leap year	(0-3)
CC	Days x 100	(0-3)
DDDDDDDD	Days	(0-66 in BCD)
HHHHHH	Hours	(0-23 in BCD)
MMMMMMM	Minutes	(0-59 in BCD)
SSSSSSS	Seconds	(0-59 in BCD)

To allow decoding of IRIG-B the clock always adds 1 second to the programmed value. Therefore, the above time must be set to the desired time minus 1 second. Leap year value should be 00 = Leap year, 01 = 1st year after leap year etc.

If the most significant 16 bits are set to LL11111111111111, the free running clock will not be loaded. The LL bits will be used to define the leap year and the clock will be forced to synchronise with an external IRIG-B source.

If the most significant 16 bits are **not** set to LL11111111111111, the free running clock will be loaded and the clock will be forced into free running mode.

Example:-

```
#include "drivers.h"
…
Error   error;

error = setClk (&card1,  0x12345678, LOAD_CLOCK);
```

8.5. readClk( )

Error readClk (MemMapping *cardHandle, Ulong *clockValue, Uword mode)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
*clockValue	Pointer to location for storing result

Description:-

The readClk ( ) function reads the current value of the local clock and saves it in the location pointed at by *clockValue. For cards using IRIG-B type clock the clock value is 64 bits. To read the clock this function is required to be called twice. The first time readClk() must be called with clockValue set to 1. The second time the value of clockValue must be set to 0. The two function calls will return the MS 32 bits followed by the LS 32 bits respectfully.

The format of the IRIG-B type clock is as follows:

1st 32 bit word: N0 CC DDDDDDD HHHHH 0000 MMMMMM SSSSSS
2nd 32 bit word: 000000 LLLLLLLLLL 00000 UUUUUUUUUUU

N = Set if card is not locked with incoming IRIG-B source
C = Days x 100
D = Days
H = Hours
M = Minutes
S = Seconds
L = Milliseconds
U = 0.5uS ticks

Example:-

```
#include "drivers.h"
…
Error   error;
Ulong   cVal;

error = readClk (&card1,  &cVal);
```

8.6. rdBCinfo( )

Error rdBCinfo (MemMapping *cardHandle, Slong msgId, AdrsInfo *info)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
msgId	Message Id of created message
*info	Pointer to structure of type AdrsInfo for saving the result

The elements of the AdrsInfo are as follows:

mdbAdd	MDB address.
rtAdd	RT table address.
lutAdd	Lookup table address
ddbAdd	DDB address
bufAdd	Data buffer address

Description:-

The rdBCinfo( ) function fills the AdrsInfo structure with the absolute card addresses of the physical elements associated with the created message.

Example:-

```
#include "drivers.h"
…
Error       error;
AdrsInfo    aInfo;

error = rdBCinfo (&card1,  1, &aInfo);
```

8.7. rdRTinfo( )

Error rdRTinfo (MemMapping *cardHandle, Sword rtNo,
Sword txrx, Sword type, Sword saMd, AdrsInfo *info);

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
RtNo	RT address
txrx	Subaddress/Mode TX/RX type – TX_TYPE or RX_TYPE
type	Subaddress type – LS_SA, HS_SA or LS_MD
saMd	Subaddress or Mode code number
*info	Pointer to structure of type AdrsInfo for saving the result

The elements of the AdrsInfo are as follows:

mdbAdd	MDB address.
rtAdd	RT table address.
lutAdd	Lookup table address
ddbAdd	DDB address
bufAdd	Data buffer address

Description:

The rdRTinfo( ) function fills the AdrsInfo structure with the absolute card addresses of the physical elements associated with the created RT subaddress or mode code.

Example:

```
#include "drivers.h"
…
Error       error;
AdrsInfo    aInfo;

Error = rdRTinfo(&card1, 1, RX_TYPE, LS_SA, 2, &aInfo);
```

8.8. rdQueue( )

Error rdQueue (MemMapping *cardHandle, Qinfo *queue);

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
queue	Pointer to Qinfo structure for saving results of function call.

The elements of the Qinfo are as follows:

qName	Name of queue type:	Q_HI_PRIORITY, Q_LO_PRIORITY, Q_STA_REPORT, Q_MESSAGE
start	Element in queue to start	
count	Number of queue elements	
action	Action to take:	Q_READ_ONLY, Q_READ_AND_CLEAR
result	Pointer to array of Uword for storing result.	

Description:

The rdQueue( ) function reads the number of queue words defined by count into the array 'result'. The queue type is defined by qName and the position within the queue is defined by 'start'. Each queue has 256 elements. It is the responsibility of the user to keep a local position pointer for tracking the current position of these cyclic queues. If the 'action' is set to Q_READ_AND_CLEAR, the elements within the queue will be cleared after being written into 'result'.

Example:

```
#include "drivers.h"
…
Error       error;
Qinfo       queue;
Uword       values[256];

queue.qName   = Q_HI_PRIORITY;
queue.start   = 0;
queue.count   = 2;
queue.action  = Q_READ_AND CLEAR;
queue.result  = values;

error = rdQueue(&card1,  &queue);
```

8.9. fileInfo( )

Error fileInfo(MemMapping *memMap, Sbyte *fName)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
*fName	Pointer to NULL terminated string defining name of file for saving the results.

Description:-

The fileInfo( ) function opens a text file of name defined by the string *fName and write table in this file defining all the created parameters and the absolute address of all the physical elements. The card is tested for which mode it is in and then the appropriate info is filled in the file. This function is intended as a debug program and is not included in any DLL files or LabWindows/CVI or LabVIEW applications.

Example:-

```
#include "drivers.h"
…
Error           error;
…
Error = fileInfo(&card1, “file.txt\0”);
```

8.10. isCardPresent( )

Error isCardPresent(MemMapping *memMap)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).

Description:-

The isCardPresent( ) function checks to see if the card is present in the system. If false then an error is returned. If the card is detected E_NO_ERROR is returned.

Example:-

```
#include "drivers.h"
…
Error       error;
…
error = isCardPresent (&card1);
```

8.11. selfTest( )

Error selfTest(MemMapping *memMap, Uword *result)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
*result	Pointer to Uword for storing the result

Description:-

The selfTest( ) function executes an on-board selftest and returns the result in *result. For details of the result word see user manual for card.

Example:-

```
#include "drivers.h"
…
Error       error;
Uword       result
…
error = selfTest (&card1, &result);
```

8.12. drvsDebug( )

void drvsDebug(MemMapping *cardHandle)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initialised by a call to setup( ).

Description:-

The drvsDebug( ) function is a visual debugger that can be used in console mode. It allows the user to display and change data within the card.

Example:-

```
#include "drivers.h"
…
drvsDebug(&card1);
```

8.13. rWord( )

Uword rWord(MemMapping *memMap, Ulong offset)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
offset	Offset WORD address from start of card

Description:-

The rWord( ) function reads a 16 bit word from the offset word address ‘offset.

NOTE:

This is the low level read function that all the driver functions use to read values from the hardware. There is no error checking carried out. The return value is the 16 bit word as read from the card.

Example:-

```
#include "drivers.h"
…
Uword           value
…
value = rWord (&card1, 0x1000L);
```

8.14. wWord( )

void wWord(MemMapping *memMap, Ulong offset, Uword value)

Parameter	Description
cardHandle	Pointer to MemMapping structure, previously initiated by a call to setup( ).
offset	Offset WORD address from start of card
value	Value to write

Description:-

The wWord( ) function writes a 16 bit word to the location at the offset word address ‘offset’.

NOTE:

This is the low level write function that all the driver functions use to write values to the hardware. There is no error checking carried out. Nothing is returned.

Example:-

```
#include "drivers.h"
…
wWord (&card1, 0x1000L, 0xAAAA);
```

9. APPENDIX

9.1. Error Messages

MNEMONIC	No.	DESCRIPTION
E_NO_ERROR	0	No error
E_INV_CARDHANDLE	10	Card handle is not valid
E_INV_INFO	11	Invalid information field
E_INV_RTNUM	12	Invalid RT number
E_NOT_BCMRT_MODE	20	Card is not in BCMRT mode
E_NOT_MRT_MODE	21	Card is not in MRT mode
E_NOT_CM_MODE	22	Card is not in MON mode
E_CARD_RUNNING	30	Card is already running
E_CARD_HALTED	31	Card is halted
E_CANNOT_ISSUE_CMD	32	Card not responding to command
E_ALLOC_CYCLEL	40	Cycle allocation error
E_ALLOC_FRAMEL	41	Frame allocation error
E_ALLOC_SYSBLOCKL	42	System block allocation error
E_ALLOC_GAPSCYAREAL	43	System block area allocation error
E_ALLOC_SAMDL	44	Sub-address/Mode allocation error
E_ALLOC_DATAGAPL	45	Data buffer allocation error
E_ALLOC_SYSGAPL	46	System gap allocation error
E_ALLOC_DATAMEM	47	Data memory allocation error
E_ALLOC_DEFAULTS	48	Default value allocation error
E_DATA_AREA_FULL	50	Data allocation has exceeded max
E_SYS_AREA_FULL	51	System allocation has exceeded max
E_INV_CARDTYPE	100	Invalid card type
E_INV_OPMODE	101	Invalid operating mode for function
E_INV_CARDADD	102	Invalid card address
E_INV_MSGID	200	Invalid message ID
E_INV_MSGDATA	201	Invalid message DATA
E_INV_CYCLEID	210	Invalid cycle ID
E_INV_CYCLEDATA	211	Invalid cycle DATA
E_CYCLE_AREA_FULL	212	Cycle allocation has exceeded max
E_INV_FRAMEID	220	Invalid frame ID
E_INV_FRAMEDATA	221	Invalid frame DATA
E_FRAME_AREA_FULL	222	Framer allocation has exceeded max
E_INV_COUNT	240	Invalid frame TX count

E_INV_SA	310	Invalid sub-address
E_INV_SATYPE	311	Invalid sub-address type
E_SA_NOT_FOUND	312	Sub-address not found
E_RT_NOT_FOUND	313	RT not found
E_INV_MD	320	Invalid mode code
E_MD_NOT_FOUND	321	Mode code not found
E_CM_RUNNING	400	Chron Mon still running
E_CM_TRIG_DEF_ERR	401	Trigger definition error
E_CM_TRIG_SEQ_ERR	402	Trigger sequence error
E_CM_NO_TRIG	403	Trigger not set-up
E_CM_SRCH_ERR	404	Invalid search parameters
E_CM_SRCH_MAX	405	Maximum search count expired
E_CM_NO_MESSAGE	406	No messages available
E_CM_STOPPED	407	Chron Mon has stopped
E_INV_CMND	500	Invalid command
E_SELFTEST_FAILED	600	Selftest has failed
E_CARD_NOT_PRESENT	601	Card is not present
E_INV_CLOCK	602	Invalid clock value
E_DE_MON_SETUP_ERR	700	Dassault special monitor set-up error
E_DE_MON_REPORT_ERR	701	Dassault buffer report error
E_VXI_INIT	800	VXI card initialisation failure
E_VXI_IO	801	VXI card IO access failure
E_VXI_SIZE	802	VXI parameter out of range
E_NO_SUITABLE_GAP	2010	No suitable gap found
E_CM_SYNTAX_ERR	2020	Chron Mon syntax error
E_DRIVER_INIT_FAILED	3000	Failed to initialise driver for Win95/NT
E_FILE_OPEN	8000	Cannot open file
E_FILE_WRITE	8001	Cannot write to file
E_FILE_READ	8002	Cannot read from file
E_WRONG_FILE_TYPE	8003	Incorrect file type for mode No.

9.2. BC application example

```
/*
BC drivers application example
*/

#include <stdio.h>
#include <stdlib.h>
#include "drivers.h"

void main( )
{

MemMapping    card1;
Sword         opMode, cardType;
Ulong         cardAddress;
Uword         info1[SZ_SetupInfo], info2[SZ_MsgInfo], info3[SZ_MsgInfo];
Slong         msg1, msg2, msg3, msg4;
Slong         cycle1, frame1;
Slong         msglist[32], cyclelist[32];
DataInfo      datainfo;
CycleInfo     cycledef;
FrameInfo     framedef;
Uword         data1[64];
Sword         i, rtNum;
Ulong         cardMem;
Error         error;


error = E_NO_ERROR;

/*
Set card VME address to 0xE00000
*/
cardMem = (Ulong) 0x0E00000;

/*
Setup the card in BCMRT mode
*/
error = setup (&card1, NULL, VME_CARD, BCMRT_MODE, cardMem);

/*
All RTs default to disabled. Simulate RT1 only.
All simulated RTs should be setup by modBcMrtRt( ) before crMsg( ) is called.
*/
rtNum = 1;

info1[RtWhatToSet] = F_RtState;
info1[RtState]     = SIMULATED_RT;

if (error == E_NO_ERROR)
        error = modBcMrtRt (&card1, rtNum, info1);
```

```
/*
Create msg1 : HS BC->RT on primary bus, 1 data block set to 0x3910
*/
info2[WhatToSet]        = F_MsgType | F_LsIMsgGap |
                                F_LsBus | F_RT1 | F_WCnt | F_HsBus;

info2[MsgType]          = BC_RT_3910;
info2[LsIMsgGap]        = 100 * 10;
info2[RT1]              = 1;
info2[SubAdrs1]         = 1;
info2[LsBus]            = PRIMARY_BUS;
info2[HsBus]            = PRIMARY_BUS;
info2[WCnt]             = 32;

datainfo.size     = 64;
datainfo.action   = NEW;
datainfo.data     = data1;

for (i=0; i<datainfo.size; i++)
        data1[i] = 0x3910;

if (!error)
        error = crMsg (&card1, &msg1, info2, &datainfo);

/*
Create msg2 : TX message to HS sub-address PRI bus,  Use default data
*/
info3[WhatToSet]        = F_MsgType | F_LsIMsgGap | F_LsBus | F_RT1 | F_WCnt;
info3[MsgType]          = TX_MSG_3910;
info2[LsIMsgGap]        = 200 * 10;
info3[RT1]              = 1;
info3[SubAdrs1]         = 26;
info3[LsBus]            = PRIMARY_BUS;
info3[WCnt]             = 1;

if (error == E_NO_ERROR)
      error = crMsg (&card1, &msg2, info3, NULL);

/*
Create msg3 : HS BC->RT on secondary bus, 1 data block set to 0x193
*/
info2[WhatToSet]        = F_MsgType | F_LsIMsgGap | F_LsBus |
                                F_RT1 | F_WCnt | F_HsBus;
info2[MsgType]          = BC_RT_3910;
info2[LsIMsgGap]        = 300 * 10;
info2[RT1]              = 1;
info2[SubAdrs1]         = 1;
info2[LsBus]            = SECONDARY_BUS;
info2[HsBus]            = SECONDARY_BUS;
info2[WCnt]             = 32;
```

```
for (i=0; i<32; i++)
        data1[i] = 0x0193;

datainfo.size    = 32;
datainfo.action  = NEW;
datainfo.data    = data1;

if (error == E_NO_ERROR)
        error = crMsg (&card1, &msg3, info2, &datainfo);

/*
Create msg4 : TX message to HS sub-address SEC bus, use default data
*/
info3[0]                = F_MsgType | F_LsIMsgGap | F_LsBus | F_RT1 | F_WCnt;
info3[MsgType]          = TX_MSG_3910;
info2[LsIMsgGap]        = 400 * 10;
info3[RT1]              = 1;
info3[SubAdrs1]         = 26;
info3[LsBus]            = SECONDARY_BUS;
info3[WCnt]             = 1;

if (error == E_NO_ERROR)
        error = crMsg (&card1, &msg4, info3, NULL);

/*
Create a cycle which includes the 4 unique messages
*/
cycledef.nmrMsgs        = 4;
cycledef.msgIdL         = msglist;
cycledef.action         = NEW;

msglist[0]      = msg1;
msglist[1]      = msg2;
msglist[2]      = msg3;
msglist[3]      = msg4;

if (error == E_NO_ERROR)
      error = crCycle (&card1, &cycle1, &cycledef);

/*
Create a frame which includes the same cycle repeated 4 times
*/
framedef.nmrCycles      = 4;
framedef.cyIdL          = cyclelist;
framedef.action         = NEW;

cyclelist[0]    = cycle1;
cyclelist[1]    = cycle1;
cyclelist[2]    = cycle1;
cyclelist[3]    = cycle1;
```

```
if (error == E_NO_ERROR)
        error = crFrame (&card1, &frame1, &framedef);

/*
Run the frame forever
*/
if (error == NO_ERROR)
         error = runFrame (&card1, frame1, FOREVER);
```

9.3. MRT application example

```
#include <stdio.h>
#include <stdlib.h>
#include "drivers.h"

void main( )
{

MemMapping    card1;
Uword         info[SZ_MrtRtInfo];
Sword         i, rtNum;
SaInfo        subAdrs;
DataInfo      data;
Uword         dataVals[96];
MdInfo        mode;
MdErrInfo     errData;
Ulong         cardMem;
Sword         cardType, opMode;
Error         error;

error = E_NO_ERROR;

/*
Set card VME address to 0xE00000
*/
cardMem = (Ulong) 0x0E00000

/*
Initialise card for MRT mode
*/
error = setup (&card1, NULL, VME_CARD, MRT_MODE, cardMem);


/*
Simulate RT 1 and disable all HS mode codes
*/

rtNum = 1;

info[WhatToSet]        = F_RtState | F_RtHsModesDis;
info[RtState]          = SIMULATED_RT;
info[RtHsModesDis]     = BOTH;

if (error == E_NO_ERROR)
        error = modMrtRt (&card1, rtNum, info);
```

```
/*
Setup RT1 HS TX SA 1 with a buffer of 96 words.
Transmit the first 64 of these data words.
*/
rtNum = 1 ;

subAdrs.sa              = 1;
subAdrs.wrap            = NO_WRAP;
subAdrs.rxWCnt          = 0;
subAdrs.txWCnt          = 64;
subAdrs.hsRiTm          = 0;
subAdrs.hsTiTm          = 0;
subAdrs.rxData          = NULL;
subAdrs.txData          = &data;
subAdrs.rxErrData       = NULL;
subAdrs.txErrData       = NULL;

data.size       = 96;
data.action     = NEW;
data.data       = dataVals;

for (i=0;i<data.size;i++)
        data.data[i] = i ;

if (error == E_NO_ERROR)
        error = crMrtRtSa(&card1, rtNum, &subAdrs, HS_SA) ;

/*
Activate the card
*/

if (error == E_NO_ERROR)
        error = actMrt (&card1);
```

9.4. CM application example

```
/*
Simple program to trigger on the first RX message to RT01 and read it into 'buffer'.
*/

#include "drivers.h"

void main(void)
{

MemMapping    card1;
Error         error;
Uword         status, trg_res, seq_res, buffer[MAX_HS_MSG_SIZE];
MsgInfo       msg;

        static Ubyte trig[ ]    =    {"T1 Pri Cmd c01 R XX XX\0"};
        static Ubyte seq[ ]     =    {"If T1 continue\n  Finish PTC 1\0"};

        error = setup (&card1, 0, VME_CARD, CM_MODE, 0xE00000);

        if (error == E_NO_ERROR)
                error = defTrigs (&card1, trig, &trg_res);

        if (error == E_NO_ERROR)
                error = defSeq (&card1, seq, &seq_res);

        if (error == E_NO_ERROR)
                error = runCm (&card1);

        /*
        Poll until card has got capture
        */
        status = CM_RUNNING;
        while(error == E_NO_ERROR  &&  status == CM_RUNNING)
                error = isCmAct (&card1,&status);

        /*
        Get TRIG message
        */
        if (error == E_NO_ERROR)
        {
                msg.size     = MAX_HS_MSG_SIZE;
                msg.msgWords = buffer;
                error = getMsg (&card1, 0, &msg);
        }
}
```