Series 2040 Test Systems

MSP User Manual

(OS-9 Configuration)

Part Number 4200-4232
Version 1.2

Table of Contents

MSP User Manual 5
- Block Diagram 6
- MULTIPLE SERIAL PROTOCOL BOARD 7
- FUNCTIONAL CALLS 8
  - SampleCall 8
  - SendSerial 9
  - RecvSerial 10
  - uartParams 11
  - ALDL 12
  - DNREQ 13
  - UART01 14
  - UART02 15
  - UART02NR 16
  - UART68 17
  - UART68NR 18
  - XDE9067 19
  - XDE9068 20
  - CONFIGPROG 21
  - SendDLCP 22
  - RecvDLCP 23
  - dlcpParams 24
  - c2alive 26
  - c2dnrqst 27
  - c2dnld 28
  - c2nstd 29
  - c2sak 30
  - c2speed 31
  - c2upld 32
  - BOOT7E9 33
  - BOOTGMP63M 34
  - BOOTSYC 35
  - BOOTLOAD 36
  - EFLM 37
  - IDFLM 38
  - RFLM16K 39
  - PFLM16K 40

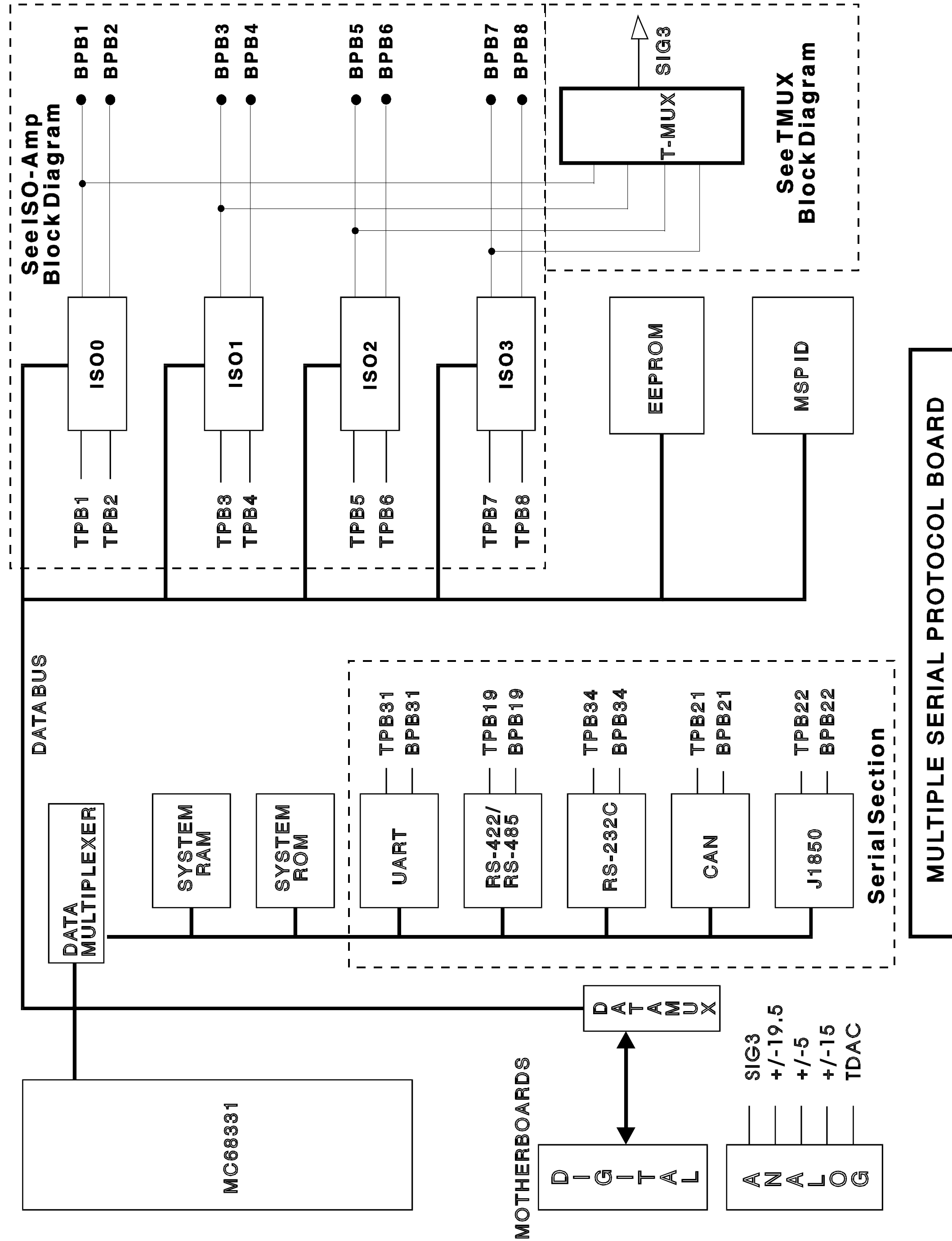
PGMFLASH 41
c2transmit 42
c2receive 43
c2transceive 44
c2disnormal 45
CanStat 46
Canrmsg 48
Canwmsg 49
Canwcontrol 51
Canwstatus 52
Canrstatus 53
Canwcpuinter 54
Canrcpuinter 55
Canwmaskshort 56
Canwmasklong 57
Canwmaskmsg15 58
Canwbittime0 59
Canwbittime1 60
Canrinterrupt 61
Canrmsgcon0 62
Canwmsgcon0 63
Canrmsgcon1 64
Canwmsgcon1 65
Canrmsgarb 66
Canwmsgarb 67
Canrmsgconfig 68
Canwmsgconfig 69
Canrmsgdata 70
Canwmsgdata 71
canConfigMsgobj 72
canErrors 73
canResetErrors 74
canPurgeBuffer 75
recvCanFrames 76
recvCanFrame15 77
sendCanFrames 78
RecvCanMsgobj 80
SendCanDataobj 81
SendCanMsgobj 82

Additional Functional Calls 83
INST 85
TMUX 87
Appendix A - MSP Error Codes 89

MSP User Manual

OS-9 Version 1.2

Block Diagram



MULTIPLE SERIAL PROTOCOL BOARD

The serial communications section of the Multiple Serial Protocol (MSP) board is designed to communicate with Units Under Test (UUTs) via a variety of serial protocols. Included are RS-232C, asynchronous RS-422/RS-485, J-1850 and Controller Area Network (CAN). Other protocols such as single wire UART lines can also be used with this card.

The MSP card has four Isolation Amplifiers (ISOAMPs). These amplifiers have differential inputs followed by a programmable gain stage, the output of which is fed through a programmable filter. The inputs of the amplifiers are “floating” and can measure small voltage differences in the presence of large common mode voltages. The Isolation Amplifiers share the functional call INST with the Instrumentation Amplifier card. This functional call is covered later in this manual.

FUNCTIONAL CALLS

The following section contains the functional calls for the MSP board. Parameters are shown for each functional call and follow this format:

SampleCall

run Sample(Param1,Param2,Param3)

WHERE:	Param1	=	1	to 1000. Time in milliseconds.
	Param2	=	0	to 300. Voltage in millivolts.
	Param3	=	0	to 300. Current in milliamps.

SendSerial

The sendSerial functional call sends a message using the default UART port on the MSP board. The message must be fully assembled by the calling program, as the function transmits the message transparently. This function may be used to send non-XDE-5024 messages. Communication is at the baud rate set up by the UARTParams call.

run sendSerial(Errcode,Smsg,Timeout)

WHERE:			
	Errcode	=	The returned error code (DIM INTEGER to get the full error code).
	Smsg	=	The message to send [DIM (n) BYTE].
	Timeout	=	The time to wait in seconds for an idle line.

EXAMPLES:

```
DIM Errcode:INTEGER
DIM Smsg(4):BYTE
Smsg(1) = $fa \Smsg(2) = $56\Smsg(3) = 5\Smsg(4) = $ab

run sendSerial(Errcode,Smsg,1) ........................................................................... send message.
```

RecvSerial

The recvSerial functional call receives a message using the default UART port on the MSP board. The message must be disassembled by the calling program as the function receives the message transparently. This function may be used to receive non-XDE-5024 messages. Communication is at the baud rate set up by the UARTParams call.

run recvSerial(Errcode,Rmsg,Timeout)

WHERE:			
	Errcode	=	The returned error code (DIM INTEGER to get the full error code).
	Rmsg	=	The message to receive [DIM (n) BYTE].
	Timeout	=	The time to wait in seconds for a message.

EXAMPLES:

```
DIM Errcode:INTEGER
DIM Rmsg(10):BYTE

run recvSerial(Errcode,Rmsg,1) ........ receive message.
```

uartParams

The uartParams functional call sets up the default parameters used by the serial functions.

```
run uartParams(paramNumber,paramValue)
```

WHERE: **paramNumber**= Index of the parameter to set.

paramValue = Value to set the parameter to.

INDEX#	PARAMETER NAME	PARAMETER VALUE
1	Port Code	0 = RS232 (Default) 1 = Single line SXR 2 = RS422
2	Baudrate	Baudrate (Default = 8192)
3	Echo Timeout	Milliseconds (Default = 10)
4	Receive Timeout	Milliseconds Default = 1000)
5	Gap Timeout	Milliseconds (Default = 15)
6	Idle Timeout	Milliseconds (Default = 1000)
8	Check Echo	0 = Do not check echo byte 1 = Check echo type (Default)

Port Code = The port to use on the MSP board
Baudrate = The baudrate to set the MSP board to
Echo Timeout = The time to wait for an echo byte.
Receive Timeout = The time to wait for a response from the sender
Gap Timeout = The maximum time allowed between characters
Idle Timeout = The maximum time to wait for an idle line before transmitting
Check Echo = Flag to compare the echoed byte with the transmitted byte

Note: The gap timeout also determines how long the MSP board will wait before determining that a generic serial message has ended.

EXAMPLES:

run uartParams(2,16384) .... Set baudrate to 16384.
run uartParams(5,1) .... set gap timeout to 1 millisecond.

ALDL

As defined in the Delco Corporate Serial Data Communications Specification, a request may be made via the serial data link to transmit product diagnostic codes and any other information specified in the product's XDE. The ALDL request is termed MODE 1. Communication is at the baud rate set up by uartParams.

```
run ALDL(Rid,Rmsg,Sid)
```

WHERE:	**Rid**	=	Receive ID or error code [DIM INTEGER to get the full error code].
	Rmsg	=	Message received from the unit under test [DIM (n) BYTE].
	Sid	=	Send ID.

DNREQ

For products that do not have illegal input conditions specified for entering the factory test mode, the Delco Corporate Serial Data Communications Specification and XDE-5024 provides a means to enter the factory test mode. This is typically referred to as a MODE 5. Once the request has been granted, RAM download messages via MODE 6 (RAM download and execute) may commence.

The DNREQ call sends a MODE 5 message to request the product enter download and execute mode. If the request is granted, a $AA is returned in Rmsg(1). Anything else signifies that the product is busy. Communication is at the baud rate set up by uartParams.

run DNREQ(Rid,Rmsg,Sid)

WHERE:			
	Rid	=	Receive ID or error code [DIM INTEGER to get the full error code].
	Rmsg	=	Message received from the unit under test [DIM (n) BYTE].
	Sid	=	Send ID.

UART01

The UART01 functional call sends a MODE 6 message to the product. When executed, the function links to the product RAM download module (temp.bin) to form a complete serial communications message. The function follows the serial protocol outlined in XDE-5024. This function downloads to product RAM starting at $0010 and communicates at the baud rate set up by uartParams.

```
run UART01(Rid,Rmsg,Sid,Smsg,)
```

WHERE:			
	Rid	=	Received ID or error code [DIM INTEGER to get the full error code.
	Rmsg	=	Message received from the unit under test [DIM (n) BYTE].
	Sid	=	Send ID.
	Smsg	=	Temp.bin index and parameters. [DIM (n) BYTE].
	Smsg(1)	=	Test number in temp.bin.
	Smsg(2)-Smsg(n)	=	Parameters for test.

UART02

The UART02 functional call sends a MODE 6 message to the product. When executed, the function links to the product RAM download module (temp.bin) to form a complete serial communication message. The function follows the serial protocol outlined in XDE-5024. Communication is at the baud rate set up by uartParams.

run UART02(Rid,Rmsg,Sid, Smsg,Ramadd)

WHERE:			
	Rid	=	Received ID error code [DIM INTEGER to get the full error code].
	Rmsg	=	Message received from the unit under test [DIM (n) BYTE].
	Sid	=	Send ID.
	Smsg	=	Temp.bin index and parameters. [Dim (n) BYTE].
	Smsg(1)	=	Test number in temp.bin.
	Smsg(2)-Smsg(n)	=	Parameters for test.
	Ramadd	=	Address to download at.

UART02NR

The UART02NR functional call sends a MODE 6 message to the product. When executed, the function links to the product RAM download module (temp.bin) to form a complete serial communication message. The function follows the serial protocol outlined in XDE-5024. This function does not get a response from the product. The function sets Rid equal to Sid if there is no communication error. Communication is at the baud rate set up by the uartParams call.

```
run UART02NR(Rid,Rmsg,Sid,Smsg,Ramadd)
```

WHERE	**Rid**	=	Receive ID or error code [DIM INTEGER to get the full error code.
	Rmsg	=	Message received from the unit under test [DIM (n) BYTE].
	Sid	=	Send ID.
	Smsg	=	Temp.bin index and parameters. [DIM (n) BYTE].
	Smsg(1)	=	Test number in temp.bin.
	Smsg(2)-Smsg(n)	=	Parameters for test.
	Ramadd	=	Address to download at.

UART68

The UART68 functional call sends a MODE 6 message to the product. When executed, the function links to the product RAM download module (temp.bin) to form a complete serial communication message. The function follows the serial protocol outlined in XDE-5024. Communication is at the baud rate set up by uartParams.

```
run UART68(Rid,Rmsg,Addr,Sid, Smsg)
```

WHERE:			
	Rid	=	Received ID error code [DIM INTEGER to get the full error code].
	Rmsg	=	Message received from the unit under test [DIM (n) BYTE].
	Addr	=	Address to download at.
	Sid	=	Send ID.
	Smsg	=	Temp.bin index and parameters. [Dim (n) BYTE].
	Smsg(1)	=	Test number in temp.bin.
	Smsg(2)-Smsg(n)	=	Parameters for test.

UART68NR

The UART68NR functional call sends a MODE 6 message to the product. When executed, the function links to the product RAM download module (temp.bin) to form a complete serial communication message. The function follows the serial protocol outlined in XDE-5024. This function does not get a response from the product. The function sets Rid equal to Sid if there is no communication error. Communication is at the baud rate set up by the uartParams call.

```
run UART02NR(Rid,Rmsg,Addr,Sid,Smsg)
```

WHERE			
	Rid	=	Receive ID or error code [DIM INTEGER to get the full error code.
	Rmsg	=	Message received from the unit under test [DIM (n) BYTE].
	Addr	=	Address to download at.
	Sid	=	Send ID.
	Smsg	=	Temp.bin index and parameters. [DIM (n) BYTE].
	Smsg(1)	=	Test number in temp.bin.
	Smsg(2)-Smsg(n)	=	Parameters for test.

XDE9067

The XDE9067 functional call is used to send and receive any type of legal XDE-5024 message. Communication is at the baud rate set up by the uartParams call.

run XDE9067(Rid,Rmsg,Timeout,Sid,{list})

WHERE:			
	Rid	=	Receive ID or error code [DIM INTEGER to get the full error code.
	Rmsg	=	Message received from unit under test [DIM (n) BYTE].
	Timeout	=	Time in seconds to wait for response from DUT. (65 seconds maximum.)
	Sid	=	Send ID.
	list	=	Any number of parameters, any size.

EXAMPLES:

DIM Rid:BYTE
DIM Rmsg(10):BYTE

run XDE9067(Rid,Rmsg,.003,$fa,5) .... Send a DNREQ

XDE9068

The XDE9068 functional call is used to send and receive any type of legal XDE-5024 message. This routine sets the baud rate of the MSP board to 16384 before the transmission starts and sets it to 8192 before it exits.

Note: This routine is provided for compatibility only. New software should set the baud rate using the uartParams call and then call XDE9067.

run XDE9068(Rid,Rmsg,Timeout,Sid,{list})

WHERE:			
	Rid	=	Receive ID or error code [DIM INTEGER to get the full error code.
	Rmsg	=	Message received from unit under test [DIM (n) BYTE].
	Timeout	=	Time in seconds to wait for response from DUT. (65 seconds maximum.)
	Sid	=	Send ID.
	list	=	Any number of parameters, any size.

EXAMPLES:

DIM Rid:BYTE
DIM Rmsg(10):BYTE

run XDE9068(Rid,Rmsg,.003,$fa,5) .... Send a DNREQ

CONFIGPROG

The CONFIGPROG functional call writes the product's HC11 configuration byte. Communication is at the baud rate set by uartParams.

run CONFIGPROG(Data,Rid)

WHERE:	**Data**	=	Byte to write to the config byte.
	Rid	=	Receive ID or error code [DIM INTEGER to get the full error code.

SendDLCP

The sendDlcp functional call sends a message using the dlcp chip on the MSP board. The message must be fully assembled by the caller as the function transmits the message transparently.

run sendDlcp(Errcode,Smsg,Timeout)

WHERE:	**Errcode**	=	The returned error code [DIM INTEGER to get the full error code.
	Smsg	=	The message to send [DIM (n) BYTE].
	Timeout	=	The time to wait in seconds for an idle line.

EXAMPLES:

```
DIM Errcode:INTEGER
DIM Smsg(4):BYTE

Smsg(1)=0 \Smsg(2)=1\Smsg(3)=2\Smsg(4)=3
run sendDlcp(Errcode,Smsg,,1) ........................................ Send message.
```

RecvDLCP

The recvDlcp functional call receives a message using the dlcp chip on the MSP board. The message must be disassembled by the caller as the function receives the message transparently.

run recvDlcp(Errcode,Rmsg,Timeout)

WHERE:			
	Errcode	=	The returned error code [DIM INTEGER to get the full error code.
	Rmsg	=	The received message [DIM (n) BYTE].
	Timeout	=	The time to wait in seconds for message.

EXAMPLES:

```
DIM Errcode:INTEGER
DIM Rmsg(12):BYTE

run recvDlcp(Errcode,Rmsg,1) ........................ Receive message.
```

dlcpParams

The dlcpParams functional call sets up default parameters used by the dlcp functions.

run dlcpParams(paramNumber, paramValue)

WHERE: **paramNumber**= Index of the parameter to set.

paramValue = Value to set the parameter to.

Index #	Parameter name	Parameter value
1	Dlcp Config	Config byte
2	Set Speed	0 = Normal speed (Default) 1 = High speed
4	Receive Timeout	Milliseconds (Default = 100)
6	Idle Timeout	Milliseconds (Default = 10)
9	Direct Command	DLCP command and data byte
10	Disable Retry	0 = Retry enabled 1 = Retry disabled (Default)
11	Reconfigure	0 = Do not configure DLCP (Default) 1 = Configure DLCP before xmit
12	Issue Reset	0 = Do not reset DLCP before xmit (Def) 1 = Reset DLCP before xmit
13	Flush Previous	0 = Do not clear RFIFO before xmit 1 = Clear RFIFO before xmit (Default)
14	Issue Flush	0 = Do not issue flush message 1 = Issue flush message before xmit (Def)
15	Check Code	0 = Do not check completion code 1 = Check completion code after xmit (Def)
16	Command	1 = Reset Transmitter 2 = Flush the firmware receive buffer

Dlcp Config	=	The config byte to write to the DLCP chip (Should not be used under normal conditions).
Set Speed	=	Command to change the speed of the DLCP chip.
Receive Timeout	=	The time to wait for a message from the product.
Idle Timeout	=	The time to wait for an idle line before transmitting.
Direct Command	=	Writes directly to the command and data registers of the DLCP chip.
Disable Retry	=	Writes terminate auto-retry command to the DLCP before each transmission.
Reconfigure	=	Write the config register of the DLCP before each transmission.
Reset	=	Writes a reset command to the DLCP before each transmission.

Flush Previous	=	Clears the RFIFO with flush byte commands before each transmission.
Issue Flush transmission.	=	Writes a flush message command to the DLCP before each
Check Comp. Code	=	Firmware waits for a completion code after a xmit and returns an error based on the completion code. This also causes the RFIFO to be flushed and an flush message command to be issued.

Notes: The Class 2 firmware is interrupt driven and interrupts should not be masked off in the dlcp chip.

A Tclear will set the parameters back to their default values.

EXAMPLES:

run dlcpParams(13,1) .... Causes the MSP firmware to clear the RFIFO .... before each transmission.
run dlcpParams(14,1) .... Causes the MSP to issue a flush message command .... before each transmission.

c2alive

The c2alive functional call sends a tool present message to the DUT to keep the 5 second activity timer reset. If the message is transmitted without error, rid will equal sid. If there was an error, the error code will be in rid. For more information, see XDE-3005.

run c2alive(Rid,Sid)

WHERE:			
	Rid	=	DUT ID if operation is error free. [DIM BYTE]
	Sid	=	ID of DUT transmitted to DUT. [DIM BYTE]

EXAMPLES:

```
DIM rid:BYTE
DIM sid:BYTE

sid = $55
Run c2alive(rid,sid)
```

c2dnrqst

The functional call c2dnrqst sends a request for block transfer message to the DUT. For more information see XDE-3005.

```
run c2dnrqst(Rid,Rmsg,Raddr,Sid,Length)
```

WHERE:

Rid	=	Received DUT ID if operation error free, error code if not. [DIM BYTE]
Rmsg	=	Data received from DUT. [DIM (n) BYTE]
Raddr	=	DUT address for block transfer. [DIM INTEGER]
Sid	=	ID of DUT transmitted to DUT. [DIM BYTE]
Length	=	Maximum data bytes for each block transfer. [DIM INTEGER]

c2dnld

The c2dnld functional call allows RAM based test modules to be downloaded to the DUT. It will also receive the response message from the DUT. For more information see XDE-3005.

run c2dnld(Rid,Rmsg,Raddr,Sid,Smsg,Timeout,Status,Mode)

WHERE:			
	Rid	=	Received DUT ID if operation error free, error code if not. [DIM BYTE]
	Rmsg	=	Data received from DUT. [DIM (n) BYTE]
	Raddr	=	DUT address to load and/or execute program. [DIM INTEGER]
	Sid	=	ID of DUT transmitted to DUT. [DIM BYTE]
	Smsg	=	TEMP.BIN test number and parameters. [DIM (n) BYTE]
	Smsg(1)	=	Test number.
	Smsg(2)-(n)	=	Parameters for test.
	Timeout	=	Maximum wait time for a ‘TOUTPUT’ response (milliseconds). [DIM INTEGER]
	Status	=	DUT response to block transfer. [Dim BYTE]
	Mode	=	Optional mode parameter. [DIM BYTE] 0 = Execute program at raddr. 1 = Download program at raddr.
	Omission	=	Download and execute at raddr.

c2nstd

The functional call c2nstd allows transmission and reception of nonstandard class 2 messages between the DUT and the functional tester. For more information see XDE-3005.

run c2nstd(Rid,Rmsg,Sid,Dataaddr,Timeout,Number)

WHERE:			
	Rid	=	Received DUT ID if operation error free, error code if not. [DIM BYTE]
	Rmsg	=	Data received from DUT. [DIM (n) BYTE]
	Sid	=	ID of DUT transmitted to DUT. [DIM BYTE]
	Dataaddr	=	Address of first data byte to be transmitted. [DIM INTEGER]
	Timeout	=	Maximum wait time in milliseconds for a 'TOUTPUT' response. [DIM INTEGER]
	Number	=	Number of data bytes to transmit. [DIM INTEGER]

c2sak

The functional call c2sak sends the seed and key messages to the DUT. For more information see XDE-3005.

run c2sak(Rid,Rmsg,Sid,Smsg,Mode)

WHERE:			
	Rid	=	Received DUT ID if operation error free, error code if not. [DIM BYTE]
	Rmsg	=	Data received from DUT. [DIM (n) BYTE]
	Rmsg(1)	=	Response code.
	Rmsg(2)	=	SEED (MSB).
	Rmsg(3)	=	SEED (LSB).
	Sid	=	ID of DUT transmitted to DUT [DIM BYTE].
	Smsg	=	Data to send to DUT [DIM (n) BYTE].
	Smsg(1)	=	KEY (MSB).
	Smsg(2)	=	KEY (LSB).
	Mode	=	Desired SEED/KEY mode [DIM BYTE]. $00 = Get SEED from DUT $FF = Send KEY to DUT

Response codes:	SEED codes
$00	= Valid acceptance of SEED request
$D8	= DUT did not accept SEED request
	KEY codes
$D8	= DUT did not accept KEY request
$33	= Product secured
$AA	= Access allowed
$CC	= Invalid key
$XX	= Any other - access denied

c2speed

The c2speed functional call changes the speed of the Class-2 bus to either normal or high speed. The call sends and receives the messages required and instructs the MSP board to change its DLCP speed. For more information see XDE-3005.

run c2speed(Rid,Rmsg,Sid,Mode)

WHERE:			
	Rid	=	Received DUT ID if operation error free, error code if not. [DIM BYTE]
	Rmsg	=	Data received from DUT.[DIM (n) BYTE]
	Rmsg(1)	=	Response code.
	Sid	=	ID of DUT transmitted to DUT. [DIM BYTE]
	Mode	=	Desired speed mode. [DIM BYTE] $01 = Normal mode $04 = High speed mode

Response codes:

$00	=	Successful speed change
$D0	=	MSP board did not respond to speed change
$D1	=	DUT did not accept disable normal communication request
$D2	=	DUT did not accept request to enter high speed mode

c2upld

The functional call c2upld requests a block of data from the DUT. For more information see XDE-3005.

run c2upld(Rid,Rmsg,Raddr,Sid,Numbytes,Upstatus)

WHERE: **Rid** = Received DUT ID if operation error free, error code if not [DIM BYTE]

Rmsg = Data received from DUT. [DIM (n) BYTE]

Raddr = DUT address for block transfer. [DIM INTEGER]

Sid = ID of DUT transmitted to DUT. [DIM BYTE]

Numbytes = Number of data bytes to be transmitted. [DIM INTEGER]

Upstatus = Response code of transfer request.

Response codes:

$FF	=	No response (generated by the MSP board)
$XX	=	DUT response

BOOT7E9

The UART FLASH programming functions are modified versions of the original Delco functions. They were modified to work with the MSP board and should operate transparently to the application test program. Please see Delco Electronics publication XDE-1041 for information on how to use these functions.

The BOOT7E9 call downloads a boot program to a GMSCM7E9 product with a 2MHz clock. The product must have been powered up in bootmode. Communication with the product is at 8192 baud and does not conform to XDE-5024.

run BOOT7E9(Errcode)

WHERE: **Errcode** = Returned error code [DIM BYTE].

BOOTGMP63M

The BOOTGMP63M call downloads a boot program to a GMP6 product with a 3MHz clock. The product must have been powered up in bootmode. Communication with the product is at 1890 baud and does not conform to XDE-5024. This function sets the baud rate of the MSP board back 8192 before exiting.

```
run BOOTGMP63M(Errcode)
```

WHERE: **Errcode** = Returned error code [DIM BYTE].

BOOTSYC

The BOOTSYC call downloads a boot program to a Sycamore product with a 4.2MHz clock. The product must have been powered up in bootmode. Communication with the product is at 8192 baud and does not conform to XDE-5024.

```
run  BOOTSYC(Errcode)
```

WHERE:	**Errcode**	=	Returned error code [DIM BYTE].

BOOTLOAD

The BOOTLOAD call downloads a boot program to an HC11 or GMP6-X product. This function is similar to the dedicated boot load functions. The bootload function accepts two additional parameters that allow it to be used with any HC11 or GMP6-X product that can be boot loaded. This function can replace the dedicated functions BOOT7E9, BOOTGMP63M, and BOOTSYC. The product must have been powered up in bootmode. Communication with the product is at the baud rate sent to the routine and does not conform to XDE-5024. The initial byte transmitted to the product sets the products baud rate. The products baud rate must within 5% of the MSP board's baudrate. The function sets the MSP baud rate to 8192 before exiting.

run BOOTLOAD(Errcode, Baudrate, BaudChar)

WHERE:	**Errcode**	=	Returned error code [DIM BYTE].
	Baudrate	=	Baudrate to set the MSP board at [DIM INTEGER]
	Baudchar	=	The initial byte to send to the product. Usually a $FF. [DIM BYTE]

EFLM

The EFLM call communicates with code bootloaded into the product to erase the products FLASH memory. This function communicates at the baud rate set up by uartParams.

run EFLM(Errcode,Commerr,Veradd,Erasestar)

WHERE:			
	Errcode	=	Returned error code for erase operation. [DIM BYTE].
	Commerr	=	Returned error code for communication error [DIM BYTE].
	Veradd	=	Verify failure at this address [DIM INTEGER].
	Erasestar	=	Starting address of the erase [DIM INTEGER].

IDFLM

The IDFLM call communicates with code bootloaded into the product to read the product's FLASH memory identification code. Communication is at the baud rate set up by uartParams.

```
run IDFLM(Errcode,Commerr,Intellid,Mode,Expectcon)
```

WHERE:	**Errcode**	=		Returned error code [DIM BYTE].
	Commerr	=		Returned error code for communication error. [DIM BYTE].
	Intellid	=		Returned intelligent ID [DIM INTEGER].
	Mode	=		Desired mode for algorithm [DIM BYTE].
		=	$DE	for read only.
		=	$DF	for read ID and program config register.
	Expectcon	=		Desired DUT config register contents [DIM BYTE].

RFLM16K

The RFLM16K call communicates with code bootloaded into the product to read the product's FLASH memory. Communication is at 16384 baud. The function sets the MSP baud rate to 8192 before exiting.

Note!! This routine will be superseded by RFLM which will communicate at the baudrate set up by a call to uartParams.

run RFLM16K(Errcode,Commerr,Startadd,Endadd,Datarray)

WHERE:			
	Errcode	=	Returned error code [DIM BYTE].
	Commerr	=	Returned error code for communication error. [DIM BYTE].
	Startadd	=	Starting address of flash read [DIM INTEGER].
	Endadd	=	Ending address of flash read [DIM INTEGER].
	Datarray	=	Receiving data array [DIM (n):BYTE where n = Endadd-Startadd+1.

PFLM16K

The PFLM16K functional call communicates with code bootloaded into the product to program the product's FLASH memory. Communication is at 16384 baud. The function sets the MSP board's baud rate to 8192 before exiting.

Note: This routine will be superseded by PFLM and PGMFLASH which will communicate at the baud rate set up by a call to uartParams.

run PFLM16K(Errcode,Commerr,VeraddnProgstar,Progend, Optver)

WHERE:				
	Errcode	=		Returned error code. [DIM BYTE]
	Commerr	=		Returned error code for communication error. [DIM BYTE]
	Veradd	=		Verify failure at this address. {DIM INTEGER]
	Progstar	=		Programming start address of the FLASH. [DIM INTEGER]
	Progend	=		Programming end address of the FLASH. [DIM INTEGER]
	Optver	=		Optional verify pass [DIM BYTE].
		=	0	for no verify.

PGMFLASH

The PGMFLASH functional call communicates with code bootloaded into the product to program the product's FLASH memory. Communication is set up by the uartParams call.

run PGMFLASH(Errcode,Commerr,Modname,Veradd, Progstar,Progend,Modbase,Optver)

WHERE:				
	Errcode	=		Returned error code. [DIM BYTE]
	Commerr	=		Returned error code for communication error. [DIM BYTE]
	Modname	=		Name of the product code data module. [string]
	Veradd	=		Verify failure at this address. [DIM INTEGER]
	Progstar	=		Programming start address of the FLASH. [DIM INTEGER]
	Progend	=		Programming end address of the FLASH. [DIM INTEGER]
	Modbase	=		Product code memory base address. [DIM BYTE]
	Optver	=		Optional verify pass. [DIM BYTE]
		=	0	for no verify.

c2transmit

The functional call c2transmit allows for free form CLASS2 data transmission to a product. Essentially, the caller's buffer is transmitted as is to the product. For more information see XDE-3005.

run c2transmit(Xmitbuff,Xmitsize,Xmiterr)

WHERE:	**Xmitbuff**	=	The buffer containing data to be transmitted to the DUT. [DIM BYTE]
	Xmitsize	=	The transmit data size (number of bytes to transmit from the buffer). [DIM INTEGER]
	Xmiterr	=	The results of the communication with the MSP and product (0 = NO ERROR) [DIM INTEGER]

c2receive

The functional call c2receive allows for free form CLASS2 data reception from a product. The routine will await the message initiation for a time specified by the caller after which time the attempt is aborted and an error is returned.
For more information see XDE-3005.

```
run  c2receive(Recvbuff,Timeout,Recverr)
```

WHERE:	**Recvbuff**	=	The buffer containing data to be received from the DUT. [DIM BYTE]
	Timeout	=	The time to wait (in milliseconds) for a message initiation from the DUT. [DIM INTEGER]
	Recverr	=	The results of the communication with the MSP and product (0 = NO ERROR). [DIM INTEGER]

c2transceive

The c2transceive functional call allows for free form CLASS2 data transmission and subsequent reception from a product. The routine will await the message initiation for a time specified by the caller after which time the attempt is aborted and an error is returned. For more information see XDE-3005.

```
run c2transceive(Xmitbuff,Xmitsize,Recvbuff,Timeout,
Commerr)
```

WHERE:	**Xmitbuff**	=	The buffer containing data to be transmitted to the DUT. [DIM BYTE]
	Xmitsize	=	The transmit data size (number of bytes to transmit from the buffer). [DIM INTEGER]
	Recvbuff	=	The buffer containing data to be received from the DUT. [DIM BYTE]
	Timeout	=	The time to wait (in milliseconds) for a message initiation from the DUT. [DIM INTEGER]
	Commerr	=	The results of the communication with the MSP and product (0 = NO ERROR). [DIM INTEGER]

c2disnormal

The functional call c2disnormal sends a MODE 28 (DISABLE NORMAL COMMUNICATIONS) message to the product via the MSP card as well as receiving the response to the request. The routine will await the response initiation for a time specified after which time the attempt is aborted and an error is returned. For more information see XDE-3005.

run c2disnormal(Modaddr,Timeout,Status,Commerr)

WHERE:			
	Modaddr	=	The MODULE ADDRESS as specified in the return message in XDE-3001B. [DIM BYTE]
	Timeout	=	The time to wait (in milliseconds) for a message initiation from the DUT. [DIM INTEGER]
	Status	=	The status of the disable request (0 = NO ERROR). [DIM BYTE]
	Commerr	=	The results of the communication with the MSP and product (0 = NO ERROR). [DIM INTEGER]

Status Codes:
$00 = Normal communications disabled
$11 = Mode not supported
$12 = Sub function not supported
$22 = Conditions not correct for request

CanStat

The canstat call reads all of the following CAN controller registers:

· Control
· Status
· CPU Interface
· Global Mask - Standard
· Global Mask - Extended
· Message 15 Mask
· Clockout
· Bus Configuration
· Bit Timing Register 0
· Bit Timing Register 1
· Interrupt

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canstat(control , status, cpuinter, maskshort, masklong, maskmsg15, clkout, busconfig, bittime, intreg)

WHERE:	**Control**	=	Data from the CAN Control Register.
	Status	=	Data from the CAN Status Register.
	Cpuinter	=	Data from the CAN CPU Interface Register.
	Maskshort	=	Data from the CAN Global Mask - Standard Registers, 2 bytes.
	Masklong	=	Data from the CAN Global Mask - Extended Registers, 4 bytes.
	Maskmsg15	=	Data from the CAN Message 15 Mask Registers, 4 bytes.
	Clkout	=	Data from the CAN Clockout Register.
	Busconfig	=	Data from the CAN Bus Configuration Register.
	Bittime0	=	Data from the CAN Bit Timing Register 0.

Bittime1	=	Data from the CAN Bit Timing Register 1.
Intreg	=	Data from the CAN Interrupt Register.

EXAMPLES:

```
DIM control:BYTE
DIM status:BYTE
DIM cpuinter:BYTE
DIM maskshort:INTEGER
DIM masklong:INTEGER
DIM maskmsg15:INTEGER
DIM clkout:BYTE
DIM busconfig:BYTE
DIM bittime0:BYTE
DIM bittime1:BYTE
DIM intreg:BYTE

run canstat (control, status, cpuinter, maskshort, masklong, maskmsg15, clkout, busconfig, bittime0,
bittime1, intreg)
```

Canrmsg

The canrmsg call reads all of the following CAN controller message object registers:

· Message Control 0
· Message Control 1
· Arbitration Registers
· Message Configuration
· Message Data

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canrmsg(msgobj, msgcon0, msgcon1, msgarb, msgconfig, msgdata)

WHERE:			
Msgobj	=	1	to 15. CAN Message object number to read.
Msgcon0	=		Data from the CAN Message Control 0 Register.
Msgcon1	=		Data from the CAN Message Control 1 Register.
Msgarb	=		Data from the CAN Message Arbitration Registers, 4 bytes.
Msgconfig	=		Data from the CAN Message Configuration Register.
Msgdata	=		Data array from the CAN Message Data Registers, 8 bytes.

EXAMPLES:

```
DIM msgcon0:BYTE
DIM msgcon1:BYTE
DIM msgarb:INTEGER
DIM msgconfig:BYTE
DIM msgdata(8):BYTE


run canrmsg (1, msgcon0, msgcon1, msgarb, msgconfig, msgdata)
```

Canwmsg

The canwmsg call writes data to the following CAN controller message object registers:

- · Message Control 0
- · Message Control 1
- · Arbitration Registers
- · Message Configuration
- · Message Data

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canwmsg(msgobj, msgcon0, msgcon1, msgarb, msgconfig, msgdata)

WHERE:	**Msgobj**	=	1	to 15. CAN Message object number to write.
	Msgcon0	=		Data to be written to the CAN Message Control 0 Register.
	Msgcon1	=		Data to be written to the CAN Message Control 1 Register.
	Msgarb	=		Data to be written to the CAN Message Arbitration Registers, 4 bytes.
	Msgconfig	=		Data to be written to the CAN Message Configuration Register.
	Msgdata	=		Data array to be written to the CAN Message Data Registers, 8 bytes

EXAMPLES:

```
DIM msgdata(8):BYTE

msgdata(0) = 0
msgdata(1) = 1
msgdata(2) = 2
msgdata(3) = 3
msgdata(4) = 4
msgdata(5) = 5
msgdata(6) = 6
msgdata(7) = 7

run canwmsg (1, $A5, $55, 0, $8C, msgdata)
```

Canwcontrol

The canwcontrol functional call writes data to the CAN Control register.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

```
run canwcontrol(Control)
```

WHERE:	Control	=	Data to be written to the CAN Control Register. [DIM BYTE]

EXAMPLES:

run canwcontrol($4A)

Canwstatus

The canwstatus functional call writes data to the CAN Status register.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canwstatus(Status)

WHERE:	**Status**	=	Data to be written to the CAN Status Register. [DIM BYTE]

EXAMPLES:

```
run canwstatus($07)
```

Canrstatus

The canrstatus functional call reads data from the CAN Status register.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canrstatus(Status)

WHERE:	**Status**	=	Data read from the CAN Status Register. [DIM BYTE]

EXAMPLES:

```
Dim status:BYTE

run canrstatus(status)
```

Canwcpuinter

The canwcpuinter functional call writes data to the CAN CPU Interface register.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canwcpuinter(Cpuinter)

WHERE:	Cpuinter	=	Data to be written to the CAN CPU Interface Register. [DIM BYTE]

EXAMPLES:

```
run canwcpuinter($40)
```

Canrcpuinter

The canrcpuinter functional call reads data from the CAN CPU Interface register.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canrcpuinter(Cpuinter)

WHERE:	**Cpuinter**	=	Data read from the CAN CPU Interface Register. [DIM BYTE]

EXAMPLES:

```
DIM cpuinter:BYTE

run canrcpuinter(cpuinter)
```

Canwmaskshort

The canwmaskshort functional call writes data to the CAN Global Mask - Standard registers.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canwmaskshort(Maskshort)

WHERE: **Maskshort** = Data to be written to the CAN Global Mask - Standard Registers, 2 bytes. [DIM INTEGER]

EXAMPLES:

```
run canwmaskshort($1234)
```

Canwmasklong

The canwmasklong functional call writes data to the CAN Global Mask - Extended registers.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

```
run canwmasklong(Masklong)
```

WHERE: **Masklong** = Data to be written to the CAN Global Mask - Extended Registers, 4 bytes. [DIM INTEGER]

EXAMPLES:

```
run canwmasklong($12345678)
```

Canwmaskmsg15

The canwmaskmsg15 functional call writes data to the CAN Message 15 Mask registers.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canwmaskmsg15(Maskmsg15)

WHERE: **Maskmsg15** = Data to be written to the CAN Message 15 Mask Registers, 4 bytes. [DIM INTEGER]

EXAMPLES:

run canwmaskmsg($12345678)

Canwbittime0

The canwbittime0 functional call writes data to the CAN Bit Timing register 0.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

```
run canwbittime0(Bittime0)
```

WHERE: **Bittime0** = Data to be written to the CAN Bit Timing Register 0. [DIM BYTE]

EXAMPLES:

run canwbittime0($44)

Canwbittime1

The canwbittime1 functional call writes data to the CAN Bit Timing register 1.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

```
run canwbittime1(Bittime1)
```

WHERE:	Bittime1	=	Data to be written to the CAN Bit Timing Register 1. [DIM BYTE]

EXAMPLES:

run canwbittime1($94)

Canrinterrupt

The canrinterrupt functional call reads data from the CAN Interrupt register.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canrinterrupt(Intreg)

WHERE:	Intreg	=	Data read from the CAN Interrupt Register. [DIM BYTE]

EXAMPLES:

```
DIM intreg:BYTE

run canrinterrupt(intreg)
```

Canrmsgcon0

The canrmsgcon0 functional call reads data from the CAN Message Control 0 register.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canrmsgcon0 (Msgobj, Msgcon0)

WHERE:				
	Msgobj	=	1	to 15. CAN Message object number to read. [DIM BYTE]
	Msgcon0	=		Data read from the CAN Message Control 0 Register. [DIM BYTE]

EXAMPLES:

DIM msgcon0:BYTE

run canrmsgcon0(1,msgcon0) .... Read message object #1's control 0 register.

Canwmsgcon0

The canwmsgcon0 functional call writes data to the CAN Message Control 0 register.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canwmsgcon0 (Msgobj, Msgcon0)

WHERE: **Msgobj** = 1 to 15. CAN Message object number to write. [DIM BYTE]

Msgcon0 = Data to write to the CAN Message Control 0 Register. [DIM BYTE]

EXAMPLES:

run canwmsgcon0(1,$95)

Canrmsgcon1

The canrmsgcon1 functional call reads data from the CAN Message Control 1 register.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canrmsgcon1(Msgobj, Msgcon1)

WHERE:	**Msgobj**	=	1	to 15. CAN Message object number to read. [DIM BYTE]
	Msgcon1	=		Data read from the CAN Message Control 1 Register. [DIM BYTE]

EXAMPLES:

DIM msgcon1:BYTE

run canrmsgcon1(1,msgcon1) .... Read message object #1's control 1 register.

Canwmsgcon1

The canwmsgcon1 functional call writes data to the CAN Message Control 1 register.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canrwmsgcon1 (Msgobj, Msgcon1)

WHERE:	Msgobj	=	1 to 15. CAN Message object number to write. [DIM BYTE]
	Msgcon1	=	Data to write to the CAN Message Control 1 Register. [DIM BYTE]

EXAMPLES:

```
run canwmsgcon1(1,$55)
```

Canrmsgarb

The canrmsgarb functional call reads data from the CAN Message Arbitration registers.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canrmsgarb(Msgobj, Msgarb)

WHERE:	**Msgobj**	=	1	to 15. CAN Message object number to read. [DIM BYTE]
	Msgarb	=		Data read from the CAN Message Arbitration Registers, 4 bytes. [DIM INTEGER]

EXAMPLES:

DIM msgarb:INTEGER

run canrmsgarb(1,msgarb) .... Read message object #1's arbitration register.

Canwmsgarb

The canwmsgarb functional call writes data to the CAN Message Arbitration registers.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canwmsgarb (Msgobj, Msgarb)

WHERE:				
	Msgobj	=	1	to 15. CAN Message object number to write. [DIM BYTE]
	Msgarb	=		Data array to be written to the CAN Message Arbitration Registers, 4 bytes. [DIM INTEGER]

EXAMPLES:

run canwmsgarb(1,$12345678)

Canrmsgconfig

The canrmsgconfig functional call reads data from the CAN Message Configuration register.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canrmsgconfig (Msgobj, Msgconfig)

WHERE:			
Msgobj	=	1	to 15. CAN Message object number to read. [DIM BYTE]
Msgconfig	=		Data read from the CAN Message Configuration Register. [DIM BYTE]

EXAMPLES:

DIM msgconfig:BYTE

run canrmsgconfig(1,msgconfig) .... Read message object #1's configuration register.

Canwmsgconfig

The canwmsgconfig functional call writes data to the CAN Message Configuration register.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canwmsgconfig (Msgobj, Msgconfig)

WHERE:	Msgobj	=	1	to 15. CAN Message object number to write. [DIM BYTE]
	Msgconfig	=		Data to write to the CAN Message Configuration Register. [DIM BYTE]

EXAMPLES:

run canwmsgconfig(1,$8C)

Canrmsgdata

The canrmsgdata functional call reads data from the CAN Message Data registers.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canrmsgdata (Msgobj, Msgdata)

WHERE:	Msgobj	=	1	to 15. CAN Message object number to read. [DIM BYTE]
	Msgdata	=		Data array read from the CAN Message Data Registers, 8 bytes. [DIM (n) BYTE]

EXAMPLES:

DIM msgdata(8):BYTE

run canrmsgdata(1,msgdata) .... Read message object #1's data registers.

Canwmsgdata

The canwmsgdata functional call writes data to the CAN Message Data registers.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run canwmsgdata (Msgobj, Msgdata)

WHERE:			
Msgobj	=	1	to 15. CAN Message object number to write. [DIM BYTE]
Msgdata	=		Data array to be written to the CAN Message Data Registers, 8 bytes. [DIM (n) BYTE]

EXAMPLES:

```
DIM msgdata(8):BYTE

msgdata(0) = $01
msgdata(1) = $23
msgdata(2) = $45
msgdata(3) = $67
msgdata(4) = $89
msgdata(5) = $AB
msgdata(6) = $CD
msgdata(7) = $EF

run canwmsgdata(1,msgdata)
```

canConfigMsgobj

The canConfigMsgobj call is a frame based function which configures a message object with the passed parameters. An error message will be returned if the Message Object is busy transmitting or receiving messages.

run canConfigMsgobj(Msgobj,Arbid,Direction,Valid)

WHERE:				
	Msgobj	=	1	to 15. The message object to configure. [DIM BYTE]
	Arbid	=		The message object arbitration ID - 4 bytes. [DIM INTEGER]
	Direction	=	0	Configure as a receive message object.
		=	1	Configure as a transmit message object. [DIM BYTE]
	Valid	=	0	Unconfigure message object by marking it invalid.
		=	1	Make message object active by marking it valid. [DIM BYTE]

EXAMPLES:

run canconfigmsgobj(2,$12345678,1,1) .... Configure message object #1 as a transmitter
.... with an arbitration ID of $12345678.

canErrors

The canErrors call retrieves the value of each CAN Controller error counter and the total number of errors.

run canErrors(errcount, spurint, msglost, msgundefined, stuff, form, ack, bit1, bit0, crc)

WHERE:			
	errcount	=	Total number of errors. [DIM INTEGER]
	spurint	=	Spurious interrupt error counter. [DIM INTEGER]
	msglost	=	Message lost error counter. [DIM INTEGER]
	msgundefined	=	Message undefined error counter. [DIM INTEGER]
	stuff	=	Stuff error counter. [DIM INTEGER]
	form	=	Form error counter. [DIM INTEGER]
	ack	=	Acknowledgment error counter. [DIM INTEGER]
	bit1	=	Bit 1 error counter. [DIM INTEGER]
	bit0	=	Bit 0 error counter. [DIM INTEGER]
	crc	=	CRC error counter. [DIM INTEGER]

EXAMPLES:

```
DIM errcount:INTEGER
DIM spurint, msglost, msgundefined:INTEGER
DIM stuff, form, ack, bit1, bit0, crc:INTEGER

run canerrors(errcount,spurint,msglost,msgundefined,stuff,form,ack,bit1,bit0,crc)
```

canResetErrors

The canResetErrors call resets the CAN controller error counters.

```
run  canResetErrors
```

EXAMPLES:

run canResetErrors

canPurgeBuffer

The canPurgeBuffer call is a frame based function which purges CAN Controller message object buffers. The message object buffer to purge will be selected based on the supplied mode and Arbitration ID parameters.

```
run canPurgeBuffer(mode,arbid)
```

WHERE:				
	mode	=		The method to use to determine which message object buffer to purge. [DIM BYTE]
		=	0	Purge all message object buffers.
		=	1	Purge a transmit message object buffer whose arbitration id matches the passed parameter.
		=	2	Purge a receive message object buffer whose arbitration id matches the passed parameter.
		=	3	Purge all transmit message object buffers.
		=	4	Purge all receive message object buffers.
	arbid	=		The message object arbitration ID - 4 bytes. Used to determine which message object buffer to purge. [DIM INTEGER]

EXAMPLES:

run canPurgeBuffer(3,0) .... Purge all message object buffers which are
.... configured as transmitters.

run canPurgeBuffer(2,$12345678) .... Purge the receive message buffers whose
.... arbitration ID is $12345678.

recvCanFrames

The recvCanFrames call is a frame based function which receives the specified number of frames from message objects 1 -14 buffers. The message object which receives the frames will be selected based on the supplied Arbitration ID. An error will be returned if a receive message object with a matching Arbitration ID is not found or could not be dynamically assigned. If the number of frames specified hasn't been received before the timeout period expires, an error will be returned. The number of frames received will also be returned in the numframes variable.

A messge object 1 - 14 frame is 9 bytes long and contains the following fields:
Message configuration - 1 Byte
Data - 8 Bytes

run recvCanFrames(arbid, numframes, frames, timeout)

WHERE:	**arbid**	=		The message object arbitration ID - 4 bytes. Used to determine which message object to receive from. [DIM INTEGER]
	numframes	=		The number of frames to receive. This variable will contain the number of frames received after the call has been made. [DIM INTEGER]
	frames	=		The frame data array to place received frames into. The array size must be at least the number of frames times 9 bytes long. Array size cannot exceed 32767 bytes. [DIM (n) BYTE]
	timeout	=	0	to 65.5 seconds. The time to wait in seconds for the specified number of frames to be received. [DIM REAL]

EXAMPLES:

DIM numframes:INTEGER
DIM frames(18):BYTE

numframes = 2
run recvCanFrames($12345678,numframes,frames,1.0) .... Receive 2 frames from the
.... message object whose arbitration ID is $12345678.

recvCanFrame15

The recvCanFrame15 call is a frame based function which receives the specified number of frames from Message Object 15. If the number of frames specified hasn't been received before the timeout period expires, a timeout error will be returned. The number of frames received will also be returned in the numframes variable.

A messge object 15 frame is 13 bytes long and contains the following fields:
Arbitration ID - 4 Bytes
Message configuration - 1 Byte
Data - 8 Bytes

run recvCanFrame15(numframes, frames, timeout)

WHERE:				
	numframes	=		The number of frames to receive. This variable will contain the number of frames received after the call has been made. [DIM INTEGER]
	frames	=		The frame data array to place received frames into. The frame array size must be at least the number of frames times 13 bytes long. Array size cannot exceed 32767 bytes. [DIM (n) BYTE]
	timeout	=	0	to 65.5 seconds. The time to wait in seconds for the specified number of frames to be received. [DIM REAL]

EXAMPLES:

DIM numframes:INTEGER
DIM frames(26):BYTE

numframes = 2
run recvCanFrame15(numframes,frames,1.0) .... Receive 2 frames from message object 15.

sendCanFrames

The sendCanFrames call is a frame based function which transmits the specified number of frames from message objects 1 -14. The message object which transmits the frames will be selected based on the supplied Arbitration ID. An error will be returned if a transmit message object with a matching Arbitration ID is not found or could not be dynamically assigned. If the number of frames specified haven't been loaded into the message object's buffer before the timeout period expires, a error will be returned. The number of frames loaded into the message object buffer will also be returned in the numframes variable.

A messge object 1 - 14 frame is 9 bytes long and contains the following fields:

- Message configuration - 1 Byte
- Data - 8 Bytes

run sendCanFrames(arbid, numframes, frames, timeout)

WHERE:				
	arbid	=		The message object arbitration ID - 4 bytes. Used to determine which message object to transmit from. [DIM INTEGER]
	numframes	=		The number of frames to transmit. This variable will contain the number of frames transferred after the call has been made. [DIM INTEGER]
	frames	=		The frame data array containing the frames to transmit. The array must be at least the number of frames times 9 bytes long. Array size cannot exceed 32767 bytes. [DIM (n) BYTE]
	timeout	=	0	to 65.5 seconds. The time to wait in seconds for the specified number of frames to be loaded into the message object buffer. [DIM REAL]

EXAMPLES:

DIM numframes:INTEGER
DIM frames(9):BYTE

numframes = 1
frames(0) = $8C - Configuration Byte
frames(1) = $01
frames(2) = $23
frames(3) = $45
frames(4) = $67
frames(5) = $89
frames(6) = $AB
frames(7) = $CD
frames(8) = $EF

run sendCanFrames($12345678,numframes,frames,1.0) .... Transmit 1 frame
.... from a message object with an arbitration ID of $12345678.

RecvCanMsgobj

The recvCanMsgobj functional call is a register based function which receives one complete message from the selected message object. An error message will be returned if a message hasn't been received before the timeout period expires.

The selected message object needs to be properly configured before using this call. The receive interrupt enable in the Message Object's Control 0 register should be disabled when using this function. Also, the direction bit in the Message Object's Configuration register needs to be set to zero to receive.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run recvCanMsgobj(msgobj, msgcon0, msgcon1, msgarb, msgconfig msgdata,timeout)

WHERE:				
	msgobj	=	1	to 15. The CAN message object number to read.
	msgcon0	=		Data from the CAN Message Control 0 Register.
	msgcon1	=		Data from the CAN Message Control 1 Register.
	msgarb	=		Data from the CAN Message Arbitration Registers, 4 bytes.
	msgconfig	=		Data from the Can Message Configuraton Register.
	msgdata	=		Data array from the CAN Message Data Registers, 8 bytes.
	timeout	=	0	to 65.5 seconds. The time to wait in seconds for a message. [DIM REAL]

EXAMPLES:

DIM msgcon0:BYTE
DIM msgcon1:BYTE
DIM msgarb:INTEGER
DIM msgconfig:BYTE
DIM msgdata(8):BYTE

Run recvCanMsgobj(1, msgcon0, msgcon1, msgarb, msgconfig, msgdata, 1.0) ....
.... Read message from message object #1.

SendCanDataobj

The sendCanDataobj functional call is a register based function call which loads new data values into the selected CAN message object's data registers and then transmits that message object. An error message will be returned if a message hasn't been successfully transmitted before the timeout period expires.

The selected message object needs to be properly configured before using this call. The transmit interrupt enable in the Message Object's Control 0 register should be disabled before using this function. Also, the direction bit in the Message Object's Configuration register needs to be set to one to transmit.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

run sendCanDataobj(msgobj,smsg(),timeout)

WHERE:				
	msgobj	=	1	to 14. The CAN Controller Message Object Number. [DIM BYTE]
	smsg()	=		The data array to transmit. Valid data array size - 1 to 8 bytes. [DIM (n) BYTE]
	timeout	=	0	to 65.5 seconds to wait for a successful transmission. [DIM REAL]

EXAMPLES:

```
DIM smsg(4):BYTE
smsg(0)=0
smsg(1)=1
smsg(2)=2
smsg(3)=3

Run sendCanDataobj(1, smsg, 1.0) ....................................................... Transmit message object #1
```

SendCanMsgobj

The sendCanMsgobj functional call is a register based function which transmits the selected CAN message object previously set up with the CAN register functions. An error message will be returned if a message hasn't been successfully transmitted before the timeout period expires.

The selected message object needs to be properly configured before using this call. The transmit interrupt enable in the Message Object's Control 0 register should be disabled. Also, the direction bit in the Message Object's Configuration register needs to be set to one to transmit.

Refer to the Intel 82527 Serial Communications Controller documentation for register information.

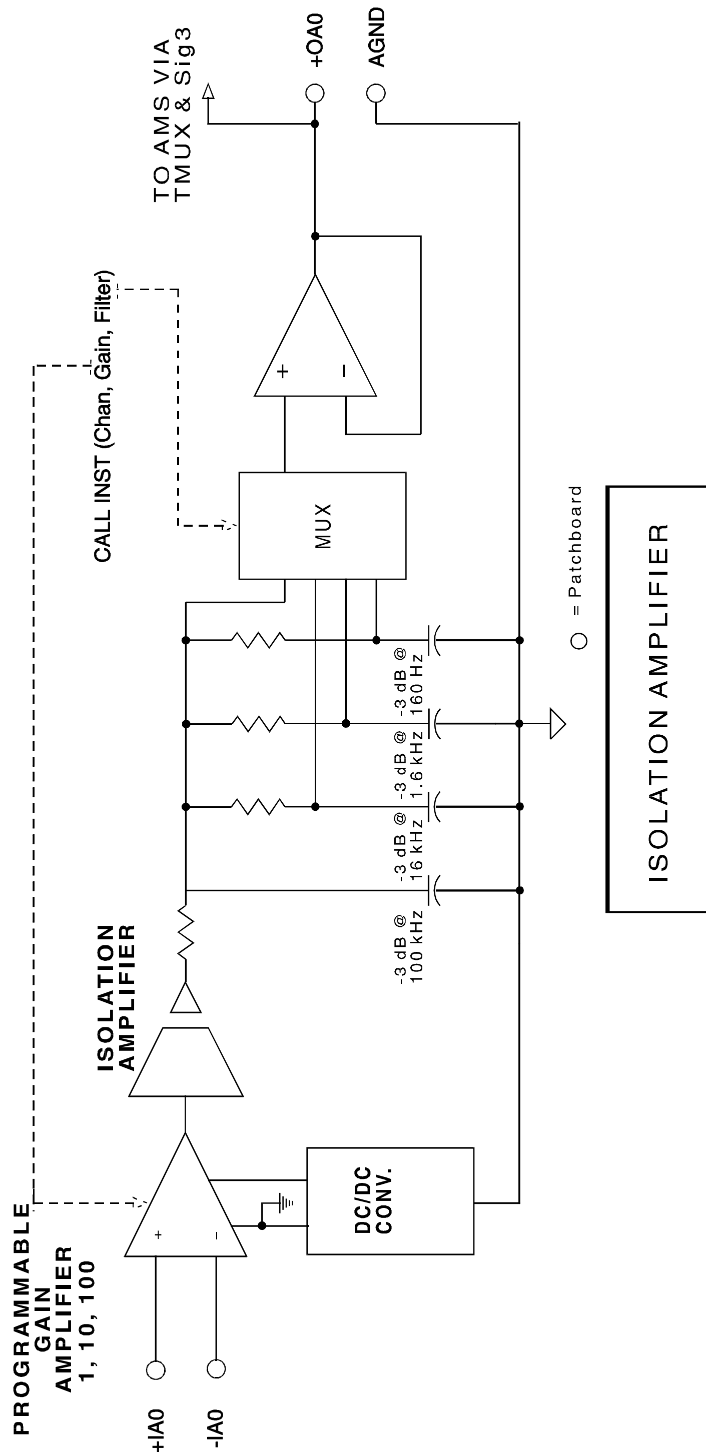
run sendCanMsgobj(msgobj, timeout)

WHERE:	**msgobj**	=	1	to 14. The CAN Controller Message Object Number. [DIM BYTE]
	timeout	=	0	to 65.5 seconds to wait for a successful transmission. [DIM REAL]

EXAMPLES:

Run sendCanMsgobj(1, 1.0) .... Transmit message object #1

ADDITIONAL FUNCTIONAL CALLS


TO AMS VIA
TMUX & Sig3
+OA0
AGND
CALL INST (Chan, Gain, Filter)
MUX
= Patchboard
-3 dB @
160 Hz
-3 dB @
1.6 kHz
-3 dB @
16 kHz
-3 dB @
100 kHz
ISOLATION AMPLIFIER
ISOLATION
AMPLIFIER
PROGRAMMABLE
GAIN
AMPLIFIER
1, 10, 100
DC/DC
CONV.
+IA0
-IA0

INST

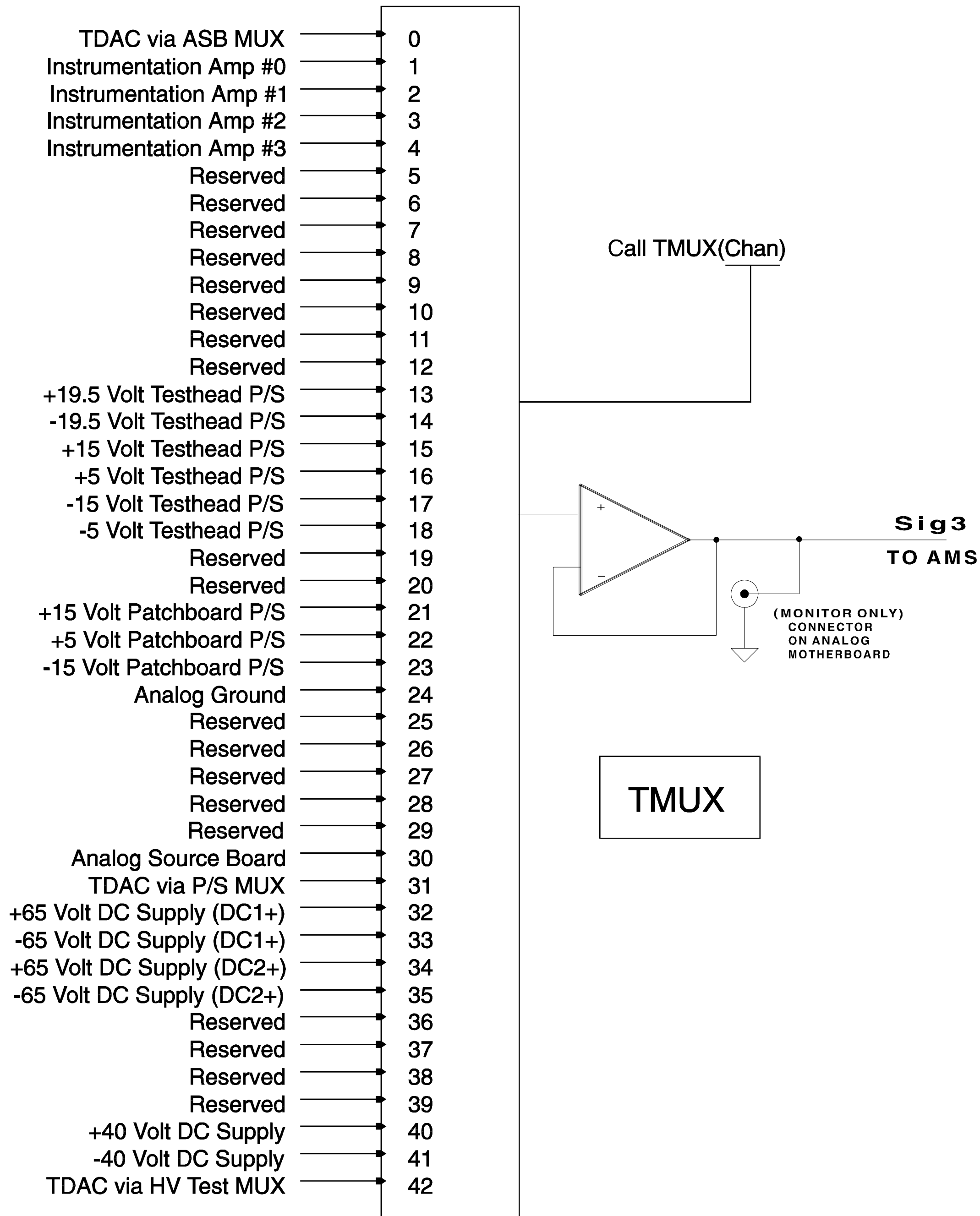
This functional call is used to set up the four differential isolation amplifiers on the MSP board. Each amplifier has programmable gain and programmable filters, and can be readback with the TMUX call.

run Inst(Chan,Gain,Filter)

WHERE:				
	Chan	=	0	to 3.
	Gain	=	0	1. (Default)
		=	1	10.
		=	2	100.
	Filter	=	0	No filter. (Default)
		=	1	16,000 Hertz, -3db (-20db/decade)
		=	2	1600 Hertz, (-20db/decade)
		=	3	160 Hertz, (-20db/decade)

EXAMPLES:

run Inst(1,2,0) .... Amplifier 1 set to Gain = 100 with no filter.
run Inst(2,1,3) .... Amplifier 2 set to gain = 10 with a 160 Hz filter.


TDAC via ASB MUX 0
Instrumentation Amp #0 1
Instrumentation Amp #1 2
Instrumentation Amp #2 3
Instrumentation Amp #3 4
Reserved 5
Reserved 6
Reserved 7
Reserved 8
Reserved 9
Reserved 10
Reserved 11
Reserved 12
+19.5 Volt Testhead P/S 13
-19.5 Volt Testhead P/S 14
+15 Volt Testhead P/S 15
+5 Volt Testhead P/S 16
-15 Volt Testhead P/S 17
-5 Volt Testhead P/S 18
Reserved 19
Reserved 20
+15 Volt Patchboard P/S 21
+5 Volt Patchboard P/S 22
-15 Volt Patchboard P/S 23
Analog Ground 24
Reserved 25
Reserved 26
Reserved 27
Reserved 28
Reserved 29
Analog Source Board 30
TDAC via P/S MUX 31
+65 Volt DC Supply (DC1+) 32
-65 Volt DC Supply (DC1+) 33
+65 Volt DC Supply (DC2+) 34
-65 Volt DC Supply (DC2+) 35
Reserved 36
Reserved 37
Reserved 38
Reserved 39
+40 Volt DC Supply 40
-40 Volt DC Supply 41
TDAC via HV Test MUX 42
Call TMUX(Chan)
Sig3
TO AMS
(MONITOR ONLY)
CONNECTOR
ON ANALOG
MOTHERBOARD
TMUX

TMUX

The Selftest Multiplexer provides readback of system signals via Sig3, which is returned to the AMS via the Analog Motherboard. It is used in calibrating the D/A's, ARB's and the AMS using TDAC as a Reference. TDAC is calibrated to a secondary standard during the Digalog Certification Procedure. TMUX is available to the USER and may be used to readback Instrumentation Amplifier outputs.

run TMux(Chan)

WHERE: **Chan**	=	0	TDAC input via Patchboard.
	=	1	Instrumentation amplifier #0.
	=	2	Instrumentation amplifier #1.
	=	3	Instrumentation amplifier #2.
	=	4	Instrumentation amplifier #3.
	=	5	Reserved.
	=	6	Reserved.
	=	7	Reserved.
	=	8	Reserved.
	=	9	Reserved.
	=	10	Reserved.
	=	11	Reserved.
	=	12	Reserved.
	=	13	+19.5 TBUS power supply.
	=	14	-19.5 TBUS power supply.
	=	15	+15 TBUS power supply.
	=	16	+5 TBUS power supply.
	=	17	-15 TBUS power supply.
	=	18	-5.2 TBUS power supply.
	=	19	Reserved.
	=	20	Reserved.
	=	21	+15PB Patchboard power supply.
	=	22	+5PB Patchboard power supply.
	=	23	-15PB Patchboard power supply.
	=	24	Analog ground.
	=	25	Reserved.
	=	26	Reserved.
	=	27	Reserved.
	=	28	Reserved.
	=	29	Reserved.
	=	30	Analog Source Board.
	=	31	TDAC via internal P/S mux.
	=	32	+65 Volt DC Supply (DC1+).

=	33	-65 Volt DC Supply (DC1-).
=	34	+65 Volt DC Supply (DC2+).
=	35	-65 Volt DC Supply (DC2-).
=	36	Reserved.
=	37	Reserved.
=	38	Reserved.
=	39	Reserved.
=	40	+40 Volt DC Supply.
=	41	-40 Volt DC Supply.
=	42	TDAC via the HV Test MUX.

EXAMPLES:

DIM Chan:INTEGER

run TMux(1) .... Multiplexes Instrumentation Amp #0 to Sig3 on the AMS.

Appendix A - MSP Error Codes

100:001 (MSP) No MSP board
100:002 (MSP) SCI port not ready
100:003 (MSP) SCI port overrun
100:004 (MSP) SCI port framing error
100:005 (MSP) SCI port noise error
100:006 (MSP) Invalid function number
100:007 (MSP) Out of memory
100:008 (MSP) MSP board is not responding to commands
100:009 (MSP) TBUS transmit timeout
100:010 (MSP) Invalid message number
100:011 (MSP) Invalid message size
100:012 (MSP) Bus error on board
100:014 (MSP) Unknown command
100:015 (MSP) MSP is already executing a command
100:020 (MSP) UART unknown message
100:021 (MSP) UART unknown parameter
100:022 (MSP) UART buffer overflow
100:025 (MSP) DLCP unknown parameter
100:030 (MSP) Exception on board
100:100 (MSP) C2DNLD Bad mode
100:101 (MSP) C2DNLD Bad test index
100:113 (MSP) C2DNLD Transfer suspended
100:114 (MSP) C2DNLD Transfer aborted
100:116 (MSP) C2DNLD Illegal address
100:117 (MSP) C2DNLD Illegal byte count
100:118 (MSP) C2DNLD Illegal block type
100:119 (MSP) C2DNLD CS error
100:120 (MSP) C2DNLD Incorrect byte count
100:190 (MSP) Mismatched echo
100:191 (MSP) Bad message length from product
100:192 (MSP) Bad checksum from product
100:193 (MSP) Timed out while waiting for response from product
100:194 (MSP) Framing, overrun, or noise error
100:195 (MSP) Timed out while waiting for an idle line
100:196 (MSP) Timed out while waiting for an echo byte
100:200 (MSP) DLCP receive FIFO invalid
100:201 (MSP) DLCP bus shorted
100:202 (MSP) DLCP timed out while waiting for an idle line
100:203 (MSP) DLCP invalid message size
100:204 (MSP) DLCP timed out while waiting for message
100:205 (MSP) DLCP timed out while waiting to transmit
100:206 (MSP) DLCP missing completion code
100:207 (MSP) DLCP completion code indicated no transmit message
100:208 (MSP) DLCP transmitter overrun

100:209 (MSP) DLCP transmitter lost arbitration
100:210 (MSP) DLCP early completion code received
100:211 (MSP) DLCP circular buffer overflow
100:212 (MSP) DLCP transmit FIFO not empty
100:230 (MSP) CAN controller in reset, sleep, or power down.
100:231 (MSP) CAN controller configuration not enabled.
100:232 (MSP) CAN register write error.
100:233 (MSP) CAN illegal controller message object.
100:234 (MSP) MSP timed out while trying to send a CAN message.
100:235 (MSP) CAN no message received.
100:236 (MSP) CAN bad parameter error.
100:237 (MSP) CAN bad array byte size.
100:238 (MSP) CAN message object’s config register bit improperly set for function called.
100:239 (MSP) CAN message object’s control register 0 message valid bit not set.
100:240 (MSP) CAN Timeout parameter is either negative or exceeds the maximum value.
100:241 (MSP) CAN controller went busoff due to errors on the CAN bus.
100:242 (MSP) CAN message object is busy transmitting messages.
100:243 (MSP) CAN message object is busy receiving messages.
100:244 (MSP) Frame size not equal to the message object buffer’s frame size.
100:245 (MSP) Message object buffer is full.
100:246 (MSP) Matching CAN message object not found.
100:247 (MSP) Invalid direction parameter.
100:248 (MSP) Invalid Message Valid parameter value.
100:249 (MSP) Timeout expired before the desired number of messages were received.
100:250 (MSP) Invalid number of frames parameter - too large or equal to zero.