February 1993

The information in this document is subject to change without notice and should not be construed as a commitment by the Massachusetts Institute of Technology. MIT assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No guarantee is made that the software is adequately or completely described in, or behaves in accordance with, this document.

Copyright ©1986 by Massachusetts Institute of Technology

All Rights Reserved.
Printed in U.S.A.

The following are trademarks of Digital Equipment Corporation:

CDD DECnet VAX
DATATRIEVE DECUS VAXcluster
DEC MicroVAX VAX Information Architecture
DEC/CMS MicroVMS VMS
DEC/MMS Rdb/VMS VT

IDL (Interactive Data Language) is copyrighted by Research Systems, Incorporated.

GRAPAC4 (Graphics Package 4) is copyrighted by Massachusetts Institute of Technology.

This document was prepared using TeX, the typesetting system developed by Donald E. Knuth at Stanford University. TeX is a registered trademark of the American Mathematical Society.
Introduction to MDS Run-Time Library Routines

The MDS Run-time Library contains procedures used throughout the MDS system for accessing MDS databases, providing program synchronization via MDS events, interpreting commands in utilities like CCL and PCL, and some general purpose routines. Many of these routines are made available for general use and provide mechanisms that can be used in application programs. For example, when you type a PCL command such as SET DATABASE MDSDATA, the PCL utility in turn calls the TDB$SET_DB routine for you. You may, in an analysis program, call TDB$SET_DB to access data directly in your program.
2 Documentation Format for Library Routines

Each RTL routine is documented using a structured format called the routine template. This section discusses the main headings in the routine template, the information that is presented under each heading, and the format used to present the information. Some or all of the following headings may appear in a routine template:

2.1 Routine name

The routine entry point name appears in bold letters at the top of the first page.

2.2 Routine overview

The routine overview appears directly below the routine name; the overview explains, usually in one or two sentences, what the routine does.

2.3 Format

The format heading follows the routine overview. The format gives the routine entry point name and the routine argument list. Optional arguments are enclosed in square brackets.
2.4 Returns

The returns section describes what information is returned to the caller as the value of the function call if the routine was called as a function instead of as a subroutine.

2.5 Arguments

The arguments heading follows the returns heading. Detailed information about each argument is provided under the arguments heading. If a routine takes no arguments, the word "None" appears.

2.6 Description

The description heading follows the arguments heading. The description section contains information about specific actions taken by the routine: interaction between routine arguments, if any; and operation of the routine within the context of VAX/VMS.

2.7 Condition Values Returned

The condition values returned section appears following the description section. It lists the condition values (typically status or completion codes that are returned by the routine).

2.8 Condition Values Signaled

The condition values signaled section lists the condition values (typically status or completion codes that may be signaled by the routine).
Example

The example heading is optional and contains a brief segment of Fortran code demonstrating the use of a routine.
3 Calling Run-Time Library Routines

You call RTL routines just like you would call any other subroutine or function. These routines can be called by any programming language as long as the language adheres to the VAX/VMS calling standards.

3.1 Return Status

If you want to check the return status of a routine you must call the routine as a function and declare the routine and the variable getting the value of the function as Integer+4 variables. If the return status is not important you can call the routine as a subroutine.

To test the return status of RTL routines in Fortran you often only need to check for TRUE (success or information return) or FALSE (error or warning return). To check for specific return conditions you must declare the particular return status (i.e., TDB$NORMAL) as an external in your routine and then compare the status returned by the RTL routine (called as a function) to the location of the particular status. To do this you must enclose the particular status in parentheses and prefix it with the %LOC keyword. For example:

```fortran
CALL TDB$SET_DB('MY_DATABASE')
IF (STATUS .EQ. %LOC(TDB$NORMAL)) THEN
.
.
.
```
3.2 Argument Passing

There are three ways to pass arguments to the routines. The method to use for any particular argument is listed in the arguments section of the routine documentation following the 'mechanism:' label.

**By reference** - This is the normal way Fortran passes byte, integer, and real variables to subroutines. In a call such as CALL MYSUB(I) where I is an integer variable, this argument is said to be passed by reference. The Fortran compiler actually passes the address of the memory location of the variable to the subroutine.

**By descriptor** - A descriptor contains two or more longwords depending on the type of descriptor used. This is the normal way Fortran passes character string variables.

**By value** - Some routines require an argument to be the actual value of a variable not the address of the variable. Fortran normally passes the address of an argument. To make Fortran pass the value, you enclose the variable in parentheses and prefix it with the keyword %VAL (I.E. CALL MYSUB(%VAL(I))).
Linking to the Run-time Library Routines

The procedure for linking to the library routines may vary from system to system depending on how the system manager has chosen to setup MDS. The library routines exist in several shared images. If these images are placed in the library file called SYSSSHARE:IMAGELIB.OLB the linker will find them automatically and no special linking instructions are necessary. Alternately, your system manager can create a separate shared image library which you can link to just as you would any object library:

$ LINK MYPROG,MDSLIB.OLB/LIBRARY

If no system-wide MDS library exists, you can link to these shared images using an linker options file, which you can create using any text editor, containing:

SYSSSHARE:MDRTL.EXE/SHARE ! For MDS$... routines
SYSSSHARE:TDBSHR.EXE/SHARE ! For TDB$... routines
SYSSSHARE:CSVSHR.EXE/SHARE ! For CSV$... routines
SYSSSHARE:MDSDCLSRH.EXE/SHARE ! For MDS/CLI utility routines

Then link with this options file using a command such as:

$ LINK MYPROG,MDSOPTIONS.OPT/OPTDSN

Since the MDS routines are in VMS shared images, there should be no need to recompile and/or link your programs between releases of the MDS software.
MDS Data Access

The MDS Run Time Library provides routines for Input and Output to MDS Databases. This chapter will discuss reading data. All of the MDS database routines have names starting with TDB$. They return Integer*4 VMS status codes and optionally print out error messages when errors are detected.
5.1 Specifying MDS Data Items

A data item in MDS is identified by three pieces of information, the name of the DATABASE, the shot identification (shot number or date and shot number), and the name of the signal.

The first thing which must be done to access any MDS data is to tell MDS the name of the database to look in. The MDS call to identify a database to use for I/O operations is TDB$SET_DB. It takes one argument which is the name of the DATABASE.

The following code fragment sets the MDS database to TESTDATA$

Integer4 TDB$SET_DB ! TDB$SET_DB function
Integer4 STATUS ! Variable to receive status

STATUS = TDB$SET_DB('TESTDATA$') ! Set to the TESTDATA$ database
IF (STATUS) THEN ! If successful then
 ELSE ! Else
END IF ! End if

The next piece of information which identifies a data item is the shot which it came from or is associated with. Depending on the way the database was configured, this is either a date and a shot number or just a shot number. A default value for the shot can be specified with a call to TDB$SET_DEFAULT or the shot specification can be put directly into the data access calls. Each database has associated with it a last shot. If no date or shot is specified then this last shot is used as the default. The last shot is generally altered by the CSV scanner. The “last” or “current” shot is the date and shot number specification of the last shot of the experiment for which data was acquired.

The final identifying piece of information for a data item is its name. TDB data names are up to 23 characters long and are case sensitive. Most signals stored by MDS utilities have upper case names. All of the interactive MDS utilities covert names to upper case before using them. The data name is always specified in the data access call itself. It is not an optional argument. The name can be prefixed by the database level identification. This prefix contains the signal character level identifier followed by a period.
There are two routines provided for reading MDS data into FORTRAN arrays. They are TDB$DATA and TDB$RAW_DATA. The first, TDB$DATA, returns an array of floating point numbers.

TDB$RAW_DATA returns whatever is stored in the database under the name requested. Except for special cases, like character data stored in the database, the routine to use is TDB$DATA.

TDB$SET_DB must be called before any calls to TDB$DATA can be done.

The following code fragment illustrates the simplest case; reading a signal from the current shot of a database.

```fortran
Real*4 BUFFER(4096) ! Buffer to hold the data
Integer*4 Samps ! Number of points returned

CALL TDB$SET_DB ('TESTDATA$') ! Use the TESTDATA$ database
CALL TDB$DATA ('MY SIGNAL', , , , 4096, BUFFER, Samps) ! Read MY SIGNAL
```

5.2 Reading Data
The next sample code fragment will attempt to read the signal from the 'current' shot. If it cannot then it will prompt for a shot number and try again until it succeeds. Note that the call to TDB$SET_ERRORS does not affect the control flow of the program it just turns on output of the error messages.

```plaintext
Integer*4 TDB$DATA          ! TDB$DATA function
Real*4   BUFFER(4096)       ! Buffer to hold the data
Integer*4 Samps             ! Number of points returned
Integer*4 SHOT/1/           ! Start with current shot

CALL TDB$SET_ERRORS(.TRUE.) ! Turn on error reporting
CALL TDB$SET_DB('TESTDATA$') ! Set database
DO WHILE(.NOT.TDB$DATA('MY SIGNAL',SHOT,,,)! Do while can't read signal
    + 4096,BUFFER,Samps)
    WRITE(*,'(A)') 'Enter Shot Number: '   ! Prompt for new shot
    READ(*,*) SHOT                           ! Read new shot number
END DO                                   ! End do

The same effect could have been achieved using a call to TDB$SET_DEFAULT and not using the shot-number argument to TDB$DATA.

This example uses that technique and reads in a raw signal which is expected to be a character string.

```plaintext
Integer*4 TDB$RAWDATA        ! TDB$RAWDATA function
Parameter BUFSSIZ = 256      ! Size of the buffer in bytes
Character*(BUFSSIZ) BUFFER   ! Buffer to hold the data
Integer*4 SHOT/1/            ! Start with the current shot

CALL TDB$SET_ERRORS(.TRUE.) ! Turn on error reporting
CALL TDB$SET_DB('TESTDATA$') ! Use the TESTDATA$ database
DO WHILE(.NOT.TDB$RAWDATA('MY SIGNAL',
    + ,,,BUFSSIZ,%REF(BUFFER))) ! Loop while can't read
    WRITE(*,'(A)') 'Enter Shot Number: '
    READ(*,*) SHOT
    CALL TDB$SET_DEFAULT(SHOT) ! Change the default shot number
END DO! End loop
```
This final example calls TDB$DATA with all of the arguments.

```plaintext
Program TEST_MDS
Implicit NONE

Integer*4 TDB$DATA ! TDB$DATA returns I*4 status

C
Real*4 BUFFER(8192) ! Buffer to hold the data
Character*63 DATABASE ! Database name to use
Character*11 DATE
Integer*4 DBLEN ! Length of the Database name
LOGICAL*1 DONE / .FALSE. / ! End index for TDB$DATA call
Integer*4 END ! Increment index for call
Integer*4 INC ! Loop index for data output
Integer*4 INDEX ! Index for points displayed
Integer*4 LABEL_INDEX ! Name of the signal to read
Integer*4 NAMLEN ! Signal name’s length
Integer*4 NUM ! Number of samples returned
Integer*4 SHOT ! Shot number
Integer*4 START ! Start index for TDB$DATA call
Character*23 SOURCE ! Source of data returned
Character*8 UNITS ! Units of data returned
```
MDS Data Access
Reading Data

Call TDB$SET_ERRORS(.TRUE.)  ! Turn on error reporting
Call LIB$GET_INPUT(DATABASE, 'Enter Database: ', DATABASE)  ! Get the name of the database
+ DBLEN
Call TDB$SET_DB(DATABASE(1:DBLEN))  ! Set the database
Do While (.NOT. DONE)  ! Loop
  Call LIB$GET_INPUT(NAME, 'Enter Signal Name: ', NAME)  ! Get the name of the signal NAMELEN)
  If (NAMELEN .EQ. 0) then
    DONE = .TRUE.  ! If no input then
  Else
    Call LIB$GET_INPUT(DATE, 'Date: ')  ! Get the date of the shot
    Write (*)'(1X,A,$)'), 'Enter Shot: '  ! Prompt for shot number
    Read (*)'(I1) SHOT  ! Read the shot number
    Write (*)'(1X,A,$)'), 'Enter Start: '  ! Prompt for starting index
    Read (*)'(I1) START  ! Read the starting index
    Write (*)'(1X,A,$)'), 'Enter End: '  ! Prompt for ending index
    Read (*)'(I1) END  ! Read the ending index
    Write (*)'(1X,A,$)'), 'Enter Increment: '  ! Prompt for increment
    Read (*)'(I1) INC  ! Read the increment
    If (TDB$DATA(NAME(1:NAMELEN), DATE, SHOT, START, END, INC, 8192, BUFFER, NUM, UNITS, SOURCE)) THEN  ! If can read the data then
      Type *, ' '  ! Type the data
      Type *, 'NAME,' From ', SOURCE,' Total of ', NUM, ' SAMPLES IN ', UNITS
      Type *, ' INDEX Value'
      LABEL_INDEX = START  ! Initialize index
      Do INDEX = 1, NUM  ! Loop through data
        Type *, LABEL_INDEX, ', ', buffer(INDEX)! Output data point
        LABEL_INDEX = LABEL_INDEX + INC  ! Increment Label
      End do
      END if  ! End if
    END IF
    End Do  ! End do
  END
  END IF  ! End if
  END
Writing Data

MDS provides two routines for putting data into MDS databases. The same set of things which identify a data item for reading must be specified when the data is written out. As above, they can either be defaulted or placed in the call which writes out the data. The two routines are TDB$PUT_DATA and TDB$PUT_LOGICAL. TDB$PUT_DATA writes an array of floating point numbers to an MDS database. TDB$PUT_LOGICAL defines a synonym for a data item. All signals written with these two routines, by default, are stored in level 'P' in the database. A prefix can be added to the signal name to force the storage to another level. Like the data access routines, TDB$SET_DB must be called before either of these routines are called.

The following code fragment writes a signal to the current shot of the MDS database called TESTDATA$.

```fortran
C ! VMS routine to get sine of angle
 Real*4 MTH$SIND
 Real*4 BUFFER(4096) ! A buffer for processed data
 Real*4 INDEX ! A loop counter

 DO INDEX = 1, 4096 ! Loop through proc array
   BUFFER(INDEX) = MTH$SIND(INDEX) ! Fill in data with sines
 END DO ! End loop

 CALL TDB$SET_DB('TESTDATA$') ! Set the database to TESTDATA$
 CALL TDB$SET_ERRORS(.TRUE.) ! Turn on error reporting
 CALL TDB$PUT_DATA('TEST_SIGNAL',,+
              4096, BUFFER, 'UNITS') ! Write the data to the current shot with units of UNITS

END
```
This next example will read in two signals, call a subroutine to process them and write out the result. It will also create a logical data item which points to the time of the first signal.

Program TEST_MDS
Implicit NONE

Integer*4 TDB$DATA ! TDB$DATA returns I*4 status

Real*4 BUFFER1(8192) ! Buffer for signal 1
Real*4 BUFFER2(8192) ! Buffer for signal 2
Character*63 DATABASE ! Name of the MDS database to use
Integer*4 DBLEN ! Length of the Database name
Integer*4 INDEX ! Loop index for data output
Integer*4 LABEL_INDEX ! Index for points displayed
Character*23 NAME1 ! Name of the first signal to read
Integer*4 NAME1LEN ! Signal 1 name's length
Character*23 NAME2 ! Name of the second signal to read
Integer*4 NAME2LEN ! Signal 2 name's length
Integer*4 NUM ! Number of samples to generate
Integer*4 NUM1 ! Number of samples for signal 1
Integer*4 NUM2 ! Number of samples for signal 2
Real*4 OUTPUT(8192) ! Buffer for processed signal
Character*20 PNAME ! Name for processed data
Integer*4 PLEN ! Length of processed name

Call TDB$SET_ERRORS($.TRUE.$) ! Turn on error reporting
Call LIB$GET_INPUT(DATABASE, 'Enter Database: ', 
   + DBLEN)
Call TDB$SET_DB(DATABASE(1:DBLEN)) ! Set the database
Call LIB$GET_INPUT(NAME1, 'Enter Signal Name: ', 
   + NAME1LEN)
If (TDB$DATA(NAME1(1:NAME1LEN)),, THEN 
   + 32768, BUFFER1, NUM1) THEN
   Call LIB$GET_INPUT(NAME2, 'Enter Signal Name: ', 
      + NAME2LEN)
   If (TDB$DATA(NAME2(1:NAME2LEN)),, THEN 
      + 32768, BUFFER2, NUM2) THEN
      NUM = JMIN0(NUM1, NUM2)
      Call PROCESS_SIGNAL(BUFFER1, BUFFER2, NUM, OUTPUT) ! Generate processed data
      Call LIB$GET_INPUT(PNAME, 'Enter Output Name: ', 
         + PLEN)
      Call TDB$PUT_DATA(PNAME(1:PLEN),,NUM,OUTPUT) ! Write the processed data out
      Call TDB$PUT_LOGICAL(PNAME(1:PLEN)//'_TM',, ! Write out the time logical
         + NAME1(1:NAME1LEN)//'_TM')
   End if
End if
End if

END
MDS Data Access
Writing Data

C Dummy analysis routine.
C NOTE this function could be done using MDS TRANSFORMS.
C

subroutine PROCESS_SIGNAL(sig1, sig2, num, out) ! Dummy analysis routine
Implicit NONE
Integer*4 num ! number of points in signals
Real*4 sig1(num) ! Input 1
Real*4 sig2(num) ! Input 2
Real*4 out(num) ! Output

C Integer*4 I ! Loop counter
C
Do I = 1, num ! Loop through data
   IF (sig2(I) .NE. 0) Then ! If second signal not zero then
      out(I) = sig1(I) + sig2(I)/sig1(I) ! compute output point
   ELSE ! Else
      out(I) = 0 ! set output point to zero
   END IF ! End if
END DO ! End do
END ! End

Documentation for all of the routines used above is in the Run-Time Library Routines section.
There are several services in the MDS Run-Time Library which provide process synchronization via the MDS event system. MDS events signify important occurrences within the operating MDS system. These events are identified by a name of from 1 to 25 characters in length. A process can inform the MDS system that it is interested in one or more events and that process will be notified when the event occurs. This notification can be via a user AST (asynchronous trap) routine or by resumption of activity if a program chooses to suspend operation until an event occurs. Likewise any process can issue events. A small buffer of data (twelve bytes) can be passed between programs issuing events and those waiting for notification.

Many MDS utilities issue and wait for events for you. For example, most CSVSCAN CAMAC module servers will issue an MDS event, if desired, after completion. Using this mechanism, a user analysis program may suspend operation until the data from a particular digitizer, for instance, has been stored in the database. Utilities like PCL and IDL can also wait for certain events to occur before attempting to plot data.

The following sections discuss the MDS Run-Time Library routines available for issuing events and receiving event notification.
6.1 Issuing MDS Events

There is only one routine necessary to issue an MDS event. That is the MDS$EVENT routine. You just call the routine specifying the name of the event you want to issue as the first argument and optionally the address of a twelve byte buffer as the second argument if you want to transmit some data with the event. You should decide on some naming conventions for events (I.E. diagnostic or users initials as prefixes) to avoid naming conflicts. The following is simple program which issues an MDS event.

```plaintext
Character*25 EVENT
Integer*4 LEN
CALL LIB$GET_INPUT(EVENT, 'Enter event name: ', LEN) ! Prompt for event name
CALL MDS$EVENT(EVENT(1:LEN)) ! Issue the event
END ! End
```

6.2 Waiting for an MDS Event

You may want your program to suspend its operation until a particular MDS event occurs. You can do this by calling the MDS$WTEVENT routine. This routine has three arguments, the name of the event you want to wait for, and optionally, a buffer (twelve bytes long) to receive any data sent by the program issuing the event, and optionally an Integer*4 variable to receive the PID (VMS process identification) of the process which issued the event. The following is a simple program that waits for the event “THE_SHOT_IS_DONE”:

```plaintext
CALL MDS$WTEVENT('THE_SHOT_IS_DONE') ! Wait for the event
END ! End
```
6.3 Queuing MDS Events

Since events are just that "events" and not states it is possible to miss an event. If you weren't at the point in your program that waits for an event before it actually occurred you would miss it. There is a mechanism in MDS to queue events which, in other words, tells MDS that you are interested in a particular event that might happen in the future. Once an event is queued, if that event happens before your program is actually waiting for it (via MDS$WTEVENT) your program won't "miss" the event. A call to MDS$WTEVENT will return immediately if the event already happened. Once queued, it is not necessary to requeue events. Each event queue is only one deep however. After you wait for an event the queue for that event is emptied. You may queue several different events concurrently. Since event queues use process local event flags it is recommended that you do not queue more than around sixteen events concurrently. You can dequeue events when you are no longer interested in them.

Events may be queued using the MDS$Q_EVENT and dequeued using the MDS$DEQ_EVENT or MDS$DEQ_ALL routines. The following is a common situation where event queueing may be usefull. A user has set up a diagnostic using two LeCroy 8212 digitizers called MY_DIGITIZER1 and MY_DIGITIZER2. He has requested event generation on the CSV form for these two modules. The digitizers are read in by a CSVSCAN CAMAC scanner and are separated in the scan list by several other large digitizer modules. The user needs to wait for the first digitizer to be read in, plot some data, wait for the second digitizer and plot some more data. Initially the user wrote his program as follows:

```
DO WHILE (.TRUE.)                         ! Do forever
   CALL MDS$WTEVENT( 'MY_DIGITIZER1' )    ! Wait for first digitizer
   CALL MY_PLOT1                          ! Plot data from the first digitizer
   CALL MDS$WTEVENT( 'MY_DIGITIZER2' )    ! Wait for second digitizer
   CALL MY_PLOT2                          ! Plot data from the second digitizer
END DO                                     ! End do
END                                         ! End
```

When running this program, the user found that he often got out the plots for the first digitizer but his program would hang until the next shot and then he would only get the second digitizer's plots. What was happening was that, depending on the load on the computer, the second digitizer was sometimes completing while he was still drawing the plots from the first digitizer. When he reached the call to wait for the second digitizer he missed the event and his program waited at this point until the next shot. The program was then changed to queue the event of the second digitizer to prevent this problem as follows:

```
CALL MDS$Q_EVENT( 'MY_DIGITIZER2' )       ! Queue the second event
DO WHILE (.TRUE.)                         ! Do forever
```

When running this program, the user found that he often got out the plots for the first digitizer but his program would hang until the next shot and then he would only get the second digitizer's plots. What was happening was that, depending on the load on the computer, the second digitizer was sometimes completing while he was still drawing the plots from the first digitizer. When he reached the call to wait for the second digitizer he missed the event and his program waited at this point until the next shot. The program was then changed to queue the event of the second digitizer to prevent this problem as follows:

```
CALL MDS$Q_EVENT( 'MY_DIGITIZER2' )       ! Queue the second event
DO WHILE (.TRUE.)                         ! Do forever
```
CALL MDS$WTEVENT( 'MY_DIGITIZER1' )! Wait for first digitizer
CALL MY_PLOT1 ! Plot data from the first digitizer
CALL MDS$WTEVENT( 'MY_DIGITIZER2' )! Wait for second digitizer
CALL MY_PLOT2 ! Plot data from the second digitizer
END DO ! End do
END ! End

The user could have queued both events but he found that he always finished the second set of plots much before the next shot and therefore it was unnecessary to queue the first event.

6.4 Establishing an MDS Event AST

There are a few run time library routines available to systems programmers for supporting the establishment of VMS asynchronous trap routines to be queued when an MDS event occurs. These routines use advanced VMS system features and are not recommended for use in user programs. The routines are MDS$EVENTAST, MDS$GETEVI, and MDS$EVENTCAN which are documented in the “Run-Time Library Routines” chapter of this manual.
Run-Time Library Routines

This part contains descriptions of the MDS Run-Time Library general-purpose routines.
CCL$DO_COMMAND

Invoke the MDS command line interpreter to execute a CCL command.

**Format**

```
CCL$DO_COMMAND command
```

**Returns**

VMS Usage: `cond_value`

- **Type:** longword (unsigned)
- **Access:** write only
- **Mechanism:** by value

**Arguments**

**Command**

VMS Usage: `char_string`

- **Type:** character string
- **Access:** read only
- **Mechanism:** by descriptor

A character string command for CCL to execute. (See the CAMAC Command Language Reference Manual for description of valid CCL commands.)

**Description**

Using the CCL$DO_COMMAND routine, it is possible to execute any of the commands available in the CCL utility from an application program. This is not a recommended method for performing CAMAC operations since it is much more efficient to call the CAMAC I/O routines directly.

This routine is contained in the shared image, SYS$LIBRARY:MDSDCLSHR.EXE, When linking your program, you must either reference this shared image in an options file or link against a shared image library containing this image.

**Condition Values Returned**

- **SS$_NORMAL**
  
  The command was successfully executed.

- **MDSDCL$_UNDEFSYM**
  
  The command issued was an unknown command or the syntax of the command was invalid.

- **SS$_xxxxxx**
  
  Any system error returned from the CAMAC I/O routines.

- **CCL$_xxxxxx**
  
  Any error returned from the CCL interface indicating improper sequence of commands or invalid values supplied for qualifiers.
SH$_{xxxxxx}$

Errors returned from CAMAC I/O indicating highway errors or invalid X and/or Q.

EXAMPLE

The following sample initializes a LeCroy 8212 module.

```
PROGRAM RESET
Call CCL$DO_COMMAND('PIO/F=06 L8212_1')    ! Enable lamps on module
CALL CCL$DO_COMMAND('SET MODULE L8212_1')  ! Set default module
Call CCL$DO_COMMAND('PIO/F=17/DATA=10010/BIN') ! SET PTS FREQ and NCHAN
Call CCL$DO_COMMAND('PIO/F=9')             ! Start scanning
Call CCL$DO_COMMAND('PIO/F=25')            ! Trigger the digitizer
END                                        ! End
```
# CSV$ASSIGN

Assign an I/O channel to a CAMAC Module for subsequent I/O operations.

## Format

<table>
<thead>
<tr>
<th>CSV$ASSIGN name, key</th>
</tr>
</thead>
</table>

## Returns

- **VMS Usage:** cond_value
- **Type:** longword (unsigned)
- **Access:** write only
- **Mechanism:** by value

## Arguments

**name**
- **VMS Usage:** logical_name
- **Type:** character string
- **Access:** read only
- **Mechanism:** by descriptor

A logical name which identifies a particular CAMAC station location at which the module to be addressed resides.

**key**
- **VMS Usage:** longword unsigned
- **Type:** longword (unsigned)
- **Access:** write only
- **Mechanism:** by reference

The address of a longword to receive the module key identifier which can be used in subsequent CAMAC I/O routines to access the camac module.

## Description

CSV$ASSIGN will call CAM$ASSIGN to assign a I/O channel to a camac module. If the assign fails the error is signaled. The condition handler of CSVSCAN will abort operation of the current module driver if the assign fails.

## Condition Value Returned

**SS$NORMAL**

The service has successfully completed.

## Condition Values Signaled

- **SS$IVDEVNAM**
  
  Invalid device name. The logical name does not reference a CAMAC device.

- **SS$IVLOGNAM**
  
  Invalid logical name. The name given was not found in the logical name table.
Run-Time Library Routines

CSV$ASSIGN

SS$_NOSUCHDEV
No such device available. The crate referenced is no available or not defined.

SS$_NOPRIV
No privilege for attempted operation. The module being referenced is restricted from general access. Your user account is not authorized to access the module.

EXAMPLE

The following is an sample program to access the CAMAC module, MY_MODULE:

```plaintext
Character*23 MODULE
Integer*4 KEY
MODULE = 'MY_MODULE'
CALL CSV$ASSIGN( MODULE , KEY )
CALL CAM$FLOW(DESCR(KEY) , 0 , 25 )
END
```

! Declare module name character string
! Declare module key
! Load module name
! Assign the module
! Perform camac F25 command on module
! END
CSV$CAMCHK

Check Camac I/O Completion Status. The CSV$CAMCHK routine will determine whether a CAMAC I/O operation completed successfully.

**FORMAT**

```plaintext
CSV$CAMCHK cam-call, [x-expected], [q-expected]
```

**RETURNS**

VMS Usage: `cond_value`
type: `longword (unsigned)`
access: `write only`
mechanism: `by value`

**ARGUMENTS**

**cam-call**
VMS Usage: `cond_value`
type: `longword (unsigned)`
access: `read only`
mechanism: `by reference`
A call to any Camac I/O routines which returns an I/O status block.

**x-expected**
VMS Usage: `longword unsigned`
type: `longword (unsigned)`
access: `read only`
mechanism: `by reference`
Optionally, the desired X state after completion of the camac I/O. If `x-expected` is set to `.TRUE.`, the call can be successful only if it returns an X. If `x-expected` is set to `.FALSE.`, the call can be successful only if it returns no X. If `x-expected` is omitted, the X is not checked.

**q-expected**
VMS Usage: `longword unsigned`
type: `longword (unsigned)`
access: `read only`
mechanism: `by reference`
Optionally, the desired Q state after completion of the camac I/O. If `q-expected` is set to `.TRUE.`, the call can be successful only if it returns a Q. If `q-expected` is set to `.FALSE.`, the call can be successful only if it returns no Q. If `q-expected` is omitted, the Q is not checked.
CSV$CAMCHK will check the return status of the Camac I/O call specified. If successful call the I/O status block of the CAMAC call is checked for success. If still successful the desired X and Q states are checked. Any error or discrepancy in X or Q desired will be signaled. The condition handler of CSVSCAN will abort operation of the current module.

**CONDITION VALUE RETURNED**

<table>
<thead>
<tr>
<th>CONDITION VALUE</th>
<th>RETURNED</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>The service has successfully completed.</td>
</tr>
</tbody>
</table>

**CONDITION VALUES SIGNALED**

- **SS$_xxxxxx** Any error returned by _cam-call_.
- **SH$_SESNOSQ** No Q received.
- **SH$_SESSQ** Q received but not expected.
- **SH$_SESNOSX** No X received.
- **SH$_SESSX** X received but not expected.
- **SH$_SCTERR** Serial transaction error.
- **SH$_SESSTE** Serial transmission error.

**EXAMPLE**

The following is an sample program to access the CAMAC module, MY_MODULE:

```plaintext
Character /*/2 /3 MODULE */ Declare module name character string
Integer/*/4 KEY */ Declare module key
Integer/*/4 CAM$PIOW /* Declare the Programmed I/O routine
MODULE = 'MY_MODULE' */ Load module name
CALL CSV$ASSIGN(MODULE, KEY) */ Assign the module
CALL CAM$CAMCHK( */ Perform camac F25 command on module
+ CAM$PIOW(%DESCR(KEY),0,25),.TRUE.,.FALSE.)
END */ END
```

MDSRTL 7-7
CSV$DO_COMMAND

Invokes the MDS command line interpreter to execute a CSV command.

**FORMAT**

<table>
<thead>
<tr>
<th>CSV$DO_COMMAND command</th>
</tr>
</thead>
</table>

**RETURNS**

VMS Usage: `cond_value`

- **type:** `longword (unsigned)`
- **access:** `write only`
- **mechanism:** `by value`

**ARGUMENTS**

VMS Usage: `char_string`

- **type:** `character string`
- **access:** `read only`
- **mechanism:** `by descriptor`

A character string command for CSV to execute. (See the Camac Server Reference Manual for description of valid CSV commands.)

**DESCRIPTION**

Using the CSV$DO_COMMAND routine, it is possible to execute any of the commands available in the CSV utility from an application program.

This routine is contained in the shared image, SYS$LIBRARY:MDSDCLSHR.EXE, When linking your program, you must either reference this shared image in an options file or link against a shared image library containing this image.

Note: The commands for manipulating MDS databases such as DIRECTOR Y, SET DATABASE, SET DEFAULT, and SET SHOT, are not included in the callable CSV. To use these commands, you must either use the TDB$DO_COMMAND routine or issue the CSV command, SET COMMAND TDB$COMMANDS, to add these database commands to the command list. Likewise, the MDS event commands will not be available unless TDB$COMMANDS is invoked.

**CONDITION VALUES RETURNED**

<table>
<thead>
<tr>
<th>SS$_NORMAL</th>
<th>The command was successfully executed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDSDCL$_UNDEFSYM</td>
<td>The command issued was an unknown command or the syntax of the command was invalid.</td>
</tr>
</tbody>
</table>
Run-Time Library Routines

CSV$DO_COMMAND

CSV$xxxxxx

Any error returned from the CSV interface indicating improper sequence of commands or invalid values supplied for qualifiers.

EXAMPLE

The following example will select a scan table, get a directory of it, and turn on one of the modules:

CALL CSV$DO_COMMAND('SET MDF MY_SCAN_TABLE') ! Select a scan table
CALL CSV$DO_COMMAND('DIRECT/MDF/FULL') ! Get a directory
CALL CSV$DO_COMMAND('MODIFY/MDF/ON MY_MODULE') ! Modify a module
END ! End
CSV$LAM_SUPPORT

Defer processing of CSV module until a LAM is generated from the specified module. This routine should only be used by CSV module drivers.

FORMAT

CSV$CAMCHK module-record, lam-module

RETURNS

VMS Usage: cond_value
type: longword (unsigned)
access: write only
mechanism: by value

ARGUMENTS

module-record
VMS Usage: mdf_record
type: longword (unsigned)
access: read only
mechanism: by reference
A module definition file record as passed to a module driver.

lam-module
VMS Usage: logical_name
type: character string
access: read only
mechanism: by descriptor
The name of a CAMAC module which is to generate a LAM whenever the module described in the module-record argument is to be processed.

DESCRIPTION

This routine should be called by module drivers which will include lam support. When called, CSV$LAM_SUPPORT will tell the CAMAC software to interrupt the scanner when a LAM is generated by the lam-module. When the LAM comes, an AST will place the module record on a to-be-processed queue and the scanner will process this module after completing the module it is currently working on.

At the end of each store and initialization pass, the scanner will check for the existence of any outstanding LAM deferrals. If any exist, the scanner will wait a period of time specified by the /TIMEOUT qualifier when the scanner was invoked. If there are outstanding LAM’s after this TIMEOUT period, an error message is generated and the requests for LAM interruption for these modules is cancelled.

CONDITION VALUE RETURNED

7-10 MDSRTL
Run-Time Library Routines
CSV$LAM_SUPPORT

**SS$_NORMAL**  
The service has successfully completed.

<table>
<thead>
<tr>
<th>CONDITION VALUES</th>
<th>SS$_xxxxxx</th>
<th>Any error returned by CAM$LAMASTW.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGNALED</td>
<td>SH$_xxxxxx</td>
<td>Any error returned in IOSB after call to CAM$LAMASTW.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

The following is a code segment from the the store routine for the L8212 module driver:

```plaintext
CALL CSV$ASSIGN(L8212.NAME, KEY)            ! Assign the camac module
CALL CSV$CAMCHR(CAM$PIOW(%DESCR(KEY),0,8),.TRUE.,) ! Check for LAM
IF (.NOT.CAM$XANDQ()) THEN
  IF (L8212.LAM_SUPPORT.EQ.'Y') THEN
    CALL CSV$LAM_SUPPORT(L8212, L8212.NAME(1:NAMLEN))! Defer till LAM
  ELSE
    CALL LIB$SIGNAL(CVS$NOT_TRIGGERED,)       ! Signal error
    END IF
  ELSE
    CALL LIB$SIGNAL(CVS$NOT_TRIGGERED,)       ! Signal error
    END IF
  END IF
ELSE
  ! process the module
```

MDSRTL 7-11
CSV$$PUT_CHANNEL

Write out data for one channel of a transient digitizer.

This routine is for use in user written CSV models. See the MDS Concepts Manual.

**FORMAT**

```
CSV$$PUT_CHANNEL shotid, module_name, channel_number, samples, bytes, dtype, buffer, channel_name, volt_vector
```

**RETURNS**

None.

**ARGUMENTS**

- **shotid**
  - VMS Usage: mds_shotid
  - Type: Longword integer (signed)
  - Access: read only
  - Mechanism: by reference
  - The shotid of the shot to write the data to. Shotid values can be generated from dates and shots, or the current shot information, by using the TDB$$SHOT_ID routine.

- **module_name**
  - VMS Usage: char_string
  - Type: character string
  - Access: read only
  - Mechanism: by descriptor
  - The name of the module the data came from. This name should be the name field from the module record.

- **channel_number**
  - VMS Usage: longword_signed
  - Type: longword (signed)
  - Access: read only
  - Mechanism: by reference
  - The address of a longword containing the channel number which the data comes from. This argument is used with the module_name argument to construct names for the raw data.
samples
VMS Usage: longword_signed
 type: longword (signed)
 access: read only
 mechanism: by reference
 The address of longword which contains the number of samples in the data
 buffer.

bytes
VMS Usage: longword_signed
 type: longword (signed)
 access: read only
 mechanism: by reference
 The address of longword which contains the number of bytes per sample in
 the data buffer.

dtype
VMS Usage: mds_data_type
 type: Byte (unsigned)
 access: read only
 mechanism: by reference
 The address of a byte which contains the data type of the raw data to be
 written. This type should be one of the defined data types (e.g. TDB$DTYTE_8BITU,...).

buffer
VMS Usage: vector
 type: unknown
 access: read only
 mechanism: by reference
 The address of a buffer of data to be written to the database. The type of
 this buffer is determined by the dtype argument. The number of elements is
determined by the samples argument.

channel_name
VMS Usage: char_string
 type: character string
 access: read only
 mechanism: by descriptor
 The address of a character string descriptor which holds the name to store the
 transformation to volts record for this channel. If this argument is blank no
 transformation to volts is stored. If it is not blank then a record with this name
 is stored. This record is a polynomial transform of a vector made from the last
 argument and the counts. In addition a logical record of channel_name_TM
 which points to the module's timing record will also be stored.
volt_vector
VMS Usage: vector_f-floating
type: F-floating
access: read only
mechanism: by reference
The address of an array of two floating point numbers. This vector holds two coefficients for a first order polynomial transformation which will convert the counts into volts.

DESCRIPTION
CSV$$PUT_CHANNEL is a utility routine for use by CSV Model support routines to store the data from the module. It implements the ‘standard’ behavior for data storage with respect to names, and transforms. This routine will always store the count data buffer with a name of module_name.nn where nn is the channel number. If the channel_name is not blank it will store the following additional records:

1 The volt_vector with a name of module_name.V.nn. This vector, supplied by the caller contains a coefficient and an offset which when applied to the counts with a polynomial transform will return volts input to the digitizer.

2 A polynomial transform named channel_name. This transform record refers to the count record and the volt_vector stored by this routine. When this record is retrieved from the database it will return the volts input to the digitizer.

3 A logical transform named channel_name_Tm. This logical transform points at the module’s timing record stored by the CSV Model support routine. This transform and the polynomial transform define a pair of names for each channel (channel_name, channel_name_Tm) which are the volts and timing for the channel.

This routine signals all errors.

CONDITION
VALUES  SIGNALED
TDB$_OPENERR Unable to open or create output file. The signaled error will contain more information as to the cause of the failure.
TDB$_PUTERR Error writing the database item. The signaled error will contain more information as to the cause of the failure.
SS$_ACCVIO Unable to reference an input argument.
EXAMPLE

The following code fragment will store one channel of data.

External TDB$K_DTYPE_12BITU ! Byte data type
Integer*4 CHANNEL ! Channel designator
Integer*4 SAMPLES ! samples to store
Integer*4 SHOTID ! TDB shot id
Real*4 VECTOR /0.0, 0.002441/ ! vector to make volts

Call TDB$SHOT_ID(,,SHOTID) ! Use default shot id

Call CSV$$PUT_CHANNEL(SHOTID, a12.NAME, CHANNEL, SAMPLES, ! write out the channel
  2, TDB$K_DTYPE_12BITU, %Val(BUFFER.ADDRESS),
  + a12.CHANNELS(CHANNEL), VECTOR)
CSV$$PUT_CHANNEL2

Write out data for one channel of a transient digitizer.

This routine is for use in user written CSV models. See the *MDS Concepts Manual*.

**FORMAT**

```
CSV$$PUT_CHANNEL2 shotid, module_name, channel_number, samples, bytes, dtype, buffer, channel_name, volt_vector
```

**RETURNS**

None.

**ARGUMENTS**

- **shotid**
  - VMS Usage: `mds_shotid`
  - type: `Longword integer (signed)`
  - access: `read only`
  - mechanism: `by reference`
  - The shotid of the shot to write the data to. Shotid values can be generated from dates and shots, or the current shot information, by using the `TDB$$SHOT_ID` routine.

- **module_name**
  - VMS Usage: `char_string`
  - type: `character string`
  - access: `read only`
  - mechanism: `by descriptor`
  - The name of the module the data came from. This name should be the name field from the module record.

- **channel_number**
  - VMS Usage: `longword Signed`
  - type: `longword (signed)`
  - access: `read only`
  - mechanism: `by reference`
  - The address of a longword containing the channel number which the data comes from. This argument is used with the module_name argument to construct names for the raw data.
samples
VMS Usage: longword_signed
        longword (signed)
type:        read only
access:        by reference
mechanism: The address of longword which contains the number of samples in the data buffer.

bytes
VMS Usage: longword_signed
        longword (signed)
type:        read only
access:        by reference
mechanism: The address of longword which contains the number of bytes per sample in the data buffer.

dtype
VMS Usage: mds_data_type
        Byte (unsigned)
type:        read only
access:        by reference
mechanism: The address of a byte which contains the data type of the raw data to be written. This type should be one of the defined data types (e.g. TDB$DTYPE_8BITU,...).

buffer
VMS Usage: vector
        unknown
type:        read only
access:        by reference
mechanism: The address of a buffer of data to be written to the database. The type of this buffer is determined by the dtype argument. The number of elements is determined by the samples argument.

channel_name
VMS Usage: char_string
        character string
type:        read only
access:        by descriptor
mechanism: The address of a character string descriptor which holds the name to store the transformation to volts record for this channel. If this argument is blank no transformation to volts is stored. If it is not blank then a record with this name is stored. This record is a polynomial transform of a vector made from the last argument and the counts. In addition a logical record of channel_name_TM which points to the module's timing record will also be stored.
Run-Time Library Routines
CSV$PUT_CHANNEL2

**vector_name**
VMS Usage: **char_string**
type: **Character string**
access: **read only**
mechanism: **by descriptor**
The address of a character string descriptor which contains the name of a data item to be used as a coefficient vector by a polynomial transform to convert counts into volts.

**DESCRIPTION**
CSV$PUT_CHANNEL2 is a utility routine for use by CSV Model support routines to store the data from the module. It implements the ‘standard’ behavior for data storage with respect to names, and transforms. This routine will always store the count data buffer with a name of module_name.nn where nn is the channel number. If the channel_name is not blank it will store the following additional records:

1 A polynomial transform named channel_name. This transform record refers to the count record stored by this routine and a coefficient vector stored by the CSV Model support routine. When this record is retrieved from the database it will return the volts input to the digitizer.

2 A logical transform named channel_name_TM. This logical transform points at the module’s timing record stored by the CSV Model support routine. This transform and the polynomial transform define a pair of names for each channel (channel_name, channel_name_TM) which are the volts and timing for the channel.

This routine signals all errors.

<table>
<thead>
<tr>
<th>CONDITION</th>
<th>VALUES</th>
<th>SIGNALED</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TDB$_OPENER</strong></td>
<td><strong>ERR</strong></td>
<td>Unable to open or create output file. The signaled error will contain more information as to the cause of the failure.</td>
</tr>
<tr>
<td><strong>TDB$_PUTERR</strong></td>
<td><strong>ERR</strong></td>
<td>Error writing the database item. The signaled error will contain more information as to the cause of the failure.</td>
</tr>
<tr>
<td><strong>SS$_ACCVIO</strong></td>
<td><strong>ERR</strong></td>
<td>Unable to reference an input argument.</td>
</tr>
</tbody>
</table>
EXAMPLE

The following code fragment will store one channel of data.

```c
External TDB$K_DTYPE_12BITU        ! Byte data type
Integer#4 CHANNEL                ! Channel designator
Integer#4 SAMPLES               ! samples to store
Integer#4 SHOTID                ! TDB shot id

C
Include 'SYS$LIBRARY:FORM$DEF($TDBDEF)'  ! TDB declarations
Character*(TDB$S_NAME) VECTOR_NAME       ! name of coeff vector
Record/TDB_DESCR/ VECTOR_DESC        ! TDB desc for Volt vector

Call TDB$SHOT_ID(,,SHOTID)         ! Use default shot id

Call TDB$PUT('S.://VECTOR_NAME, VECTOR_DESC)
Call CSV$PUT_CHANNEL2(SHOTID, a12.NAME, CHANNEL, SAMPLES,  ! write out the channel
                  2, TDB$K_DTYPE_12BITU, %Val(BUFFER.ADDRESS),
                  + 2, a12.CHANNELS(CHANNEL), VECTOR_NAME)
```

MDSRTL 7-19
CSV$$PUT_CLOCK

Write out a CLOCK record for a module.

This routine is for use in user written CSV models. See the MDS Concepts Manual.

**FORMAT**

```
CSV$$PUT_CLOCK shotid, module_name, delta_t
```

**RETURNS**

None.

**ARGUMENTS**

- **shotid**
  - VMS Usage: mds_shotid
  - type: Longword integer (signed)
  - access: read only
  - mechanism: by reference
  - The shotid of the shot to write the data to. Shotid values can be generated from dates and shots, or the current shot information, by using the TDB$$SHOT_ID routine.

- **module_name**
  - VMS Usage: char_string
  - type: character string
  - access: read only
  - mechanism: by descriptor
  - The name of the module the data came from. This name should be the name field from the module record.

- **delta_t**
  - VMS Usage: floating_point
  - type: F_floating
  - access: read only
  - mechanism: by reference
  - The address of a real*4 number which contains the delta time of the samples taken by the digitizer.
CSV$$PUT_CLOCK is a utility routine for use by CSV Model support routines to store a CLOCK record. The following is the actual layout of the fields in a CLOCK record:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELTA-TIME</td>
<td>0 to 20 Frequency Shifts</td>
</tr>
<tr>
<td>TRIGGER</td>
<td>(23 bytes)</td>
</tr>
<tr>
<td>TRIGIDX</td>
<td>24</td>
</tr>
<tr>
<td>TIME-OFFSET</td>
<td>28</td>
</tr>
<tr>
<td>NEW-DELTA-TIME</td>
<td>32</td>
</tr>
</tbody>
</table>

This routine signals all errors.

**Condition Values Signaled**

- **TDB$$OPENERR**: Unable to open or create output file. The signaled error will contain more information as to the cause of the failure.
- **TDB$$PUTERR**: Error writing the database item. The signaled error will contain more information as to the cause of the failure.
Run-Time Library Routines
CSV$PUT_CLOCK

SS$ACCVIO
Unable to reference an input argument.

EXAMPLE

The following code fragment will store a clock record.

```
Integer*4 SHOTID ! Current Shot id
Real*4  FREQ  ! Digitizer frequency

Call TDB$SHOT_ID(,,SHOTID)
Call CSV$PUT_CLOCK(shotID, a12.NAME, FREQ)
```

7-22 MDSRTL
CSV$$PUT_TIMING

Write out a timing record for a module.

This routine is for use in user written CSV models. See the MDS Concepts Manual.

**FORMAT**

```
CSV$$PUT_TIMING shotid, clock, trigger, trig_sample, samples, module_name
```

**RETURNS**

```
None.
```

**ARGUMENTS**

`shotid`
VMS Usage: `mds_shotid`
type: Longword integer (signed)
access: read only
mechanism: by reference
The shotid of the shot to write the data to. Shotid values can be generated from dates and shots, or the current shot information, by using the TDB$$SHOT_ID routine.

`clock`
VMS Usage: `char_string`
type: Character string
access: read only
mechanism: by descriptor
The address of a character string descriptor which contains the name of a CLOCK record this ATD_TIMING record should refer to.

`trigger`
VMS Usage: `char_string`
type: Character string
access: read only
mechanism: by descriptor
The address of a character string descriptor which contains the name of a TRIGGER record this ATD_TIMING record should refer to.
Run-Time Library Routines
CSV$PUT_TIMING

**trig_sample**
VMS Usage: `longword_signed`
type: `longword (signed)`
access: `read only`
mechanism: `by reference`
The address of a longword containing the sample number of the TRIGGER which corresponds to the trigger of the module. In most cases this will be 1.

**samples**
VMS Usage: `longword_signed`
type: `longword (signed)`
access: `read only`
mechanism: `by reference`
The address of longword which contains the maximum number of samples which will be stored for each channel of the digitizer.

**module_name**
VMS Usage: `char_string`
type: `character string`
access: `read only`
mechanism: `by descriptor`
The name of the module the data came from. This name should be the name field from the module record.
**DESCRIPTION**

CSV\$PUT\_TIMING is a utility routine for use by CSV Model support routines to store an ATD\_TIMING record. The following is the actual layout of the fields in an digitizer timing record:

![Diagram of timing record layout]

This routine signals all errors.

<table>
<thead>
<tr>
<th>CONDITION VALUES</th>
<th>TDB$_OPENERR</th>
<th>TDB$_PUTERR</th>
<th>SS$_ACCVIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGNALED</td>
<td>Unable to open or create output file. The signaled error will contain more information as to the cause of the failure.</td>
<td>Error writing the database item. The signaled error will contain more information as to the cause of the failure.</td>
<td>Unable to reference an input argument.</td>
</tr>
</tbody>
</table>
The following code fragment will store a timing record.

```plaintext
Integer*4 SHOTID ! Current Shot id
Character*(TDB$$_NAME) CLOCK ! Name of clock record
Character*(TDB$$_NAME) TRIGGER ! Name of trigger record

Call TDB$SHOT_ID(,,SHOTID)
CLOCK = a12.$NAME(1:LEN)//'_CK'
TRIGGER = a12.TRIGGER
Call CSV$$PUT_TIMING(SHOTID, CLOCK, TRIGGER, 1, 8192, a12.$NAME)
```

---

**EXAMPLE**
CSV$$PUT_VECTOR

Write out a CLOCK record for a module.

This routine is for use in user written CSV models. See the MDS Concepts Manual.

**FORMAT**

```
CSV$$PUT_VECTOR shotid, vector, name, module_name
```

**RETURNS**

None.

**ARGUMENTS**

- **shotid**
  - VMS Usage: `mds_shotid`
  - type: `Longword integer (signed)`
  - access: `read only`
  - mechanism: `by reference`
  - The shotid of the shot to write the data to. Shotid values can be generated from dates and shots, or the current shot information, by using the TDB$$SHOT_ID routine.

- **vector**
  - VMS Usage: `Vector_floating_point`
  - type: `F_floating array`
  - access: `read only`
  - mechanism: `by reference`
  - The address of an array of two F_floating numbers to write to the database. This array contains the Coefficients for a 1st order polynomial transform.

- **name**
  - VMS Usage: `char_string`
  - type: `character string`
  - access: `read only`
  - mechanism: `by descriptor`
  - The address of a character descriptor which contains the name to store the data under.

- **module_name**
  - VMS Usage: `char_string`
  - type: `character string`
  - access: `read only`
  - mechanism: `by descriptor`
  - The name of the module the data came from. This name should be the `name` field from the module record.
CSV$$PUT_VECTOR is a utility routine for use by CSV Model support routines to store a two element floating point array. It is used by model support routines to store the coefficients for the polynomial transformation from counts to volts.

This routine signals all errors.

**CONDITION**

**VALUES**

**SIGNALED**

| TDB$$OPENERR | Unable to open or create output file. The signaled error will contain more information as to the cause of the failure. |
| TDB$$PUTERR | Error writing the database item. The signaled error will contain more information as to the cause of the failure. |
| SS$$ACCVIO | Unable to reference an input argument. |

**EXAMPLE**

The following code fragment will store a clock record.

```plaintext
Integer*4 SHOTID
Real*4 VECTOR(2)/ 1.0, 5.0/
Character*(TDB$$_NAME) VNAME

VNAME = mod.NAME(1:LEN)//.VV
Call TDB$$SHOT_ID(,,SHOTID)
CALL CSV$$PUT_VECTOR(SHOTID, VECTOR, VNAME,
   +                      mod.NAME)
```

MDSRTL
## MDS$DEQ_ALL

### De-queue interest in all MDS Events

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>MDS$DEQ_ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETURNS</td>
<td>VMS Usage: cond_value&lt;br&gt;type: longword (unsigned)&lt;br&gt;access: write only&lt;br&gt;mechanism: by value</td>
</tr>
</tbody>
</table>

### ARGUMENTS
None.

### DESCRIPTION
MDS$DEQ_ALL will remove all of the currently queued up MDS events from an images event queue. Programs wishing to wait for events, which are not sure that they will be waiting when the events occur, can queue them with calls to MDS$Q_EVENT. This call, MDS$DEQ_ALL, removes all of the events from the queue.

If a call to MDS$Q_EVENT fails with a status MDS$NOEVENTS this routine must be called to free up the resources used by Qed events.

### CONDITION VALUE RETURNED
None. This service does not return any status.
EXAMPLE

This example queue's some events, waits for them, and then de-queue's them.

```
Integer*4 STATUS
Integer*4 MDS$DEQ_EVENT
Integer*4 MDS$Q_EVENT
Integer*4 MDS$WTEVENT
.
.
.
CALL MDS$Q_EVENT('EVENT_1')
CALL MDS$Q_EVENT('EVENT_2')
CALL MDS$Q_EVENT('EVENT_3')
CALL MDS$Q_EVENT('EVENT_4')
.
.
.
CALL MDS$WTEVENT('EVENT_1')
.
.
.
CALL MDS$DEQ_ALL
```
### MDS$DEQ_EVENT

Deque an MDS Event.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>MDS$DEQ_EVENT event</th>
</tr>
</thead>
</table>

| RETURNS | | |
|---------|----------------------|
| VMS Usage: | cond_value |
| type: | longword (unsigned) |
| access: | write only |
| mechanism: | by value |

<table>
<thead>
<tr>
<th>ARGUMENTS</th>
<th>event</th>
</tr>
</thead>
</table>

| VMS Usage: | mds_event |
| type: | character string |
| access: | read only |
| mechanism: | by descriptor |

The event name to dequeue. Must have been previously queued by a called to MDS$Q_EVENT.

| DESCRIPTION | MDS$DEQ_EVENT dequeues an event that was queued by a call to the MDS$Q_EVENT routine. |

| CONDITION VALUE RETURNED | None. |

This service does not return any status.
EXAMPLE

This example queue's some events, waits for them, and then de-queue's them.

CALL MDS$Q_EVENT('EVENT_1')
CALL MDS$WTEVENT('EVENT_1')
CALL MDS$DEQ_EVENT('EVENT_1')
MDS$EVENT

Issue an MDS System Event

**FORMAT**

MDS$EVENT event-name [,data-buffer])

**RETURNS**

VMS Usage: cond_value
type: longword (unsigned)
access: write only
mechanism: by value

**ARGUMENTS**

event-name
VMS Usage: mds_event
type: character string
access: read only
mechanism: by descriptor
The name of the event. Only the first 25 characters are used. There is no differentiation between upper and lower characters in an event name. Use alphanumeric names.

data-buffer
unsigned longword longword (unsigned) read only by reference An optional data buffer of twelve bytes. This data can optionally be read by any other process waiting for the event. This provides a mechanism for passing a small amount of information along with the event.

**DESCRIPTION**

MDS$EVENT will cause an MDS Event to occur. These events can be used to synchronize other operations taking place in the system. User programs can be written to wait for certain MDS events to take place. The Camac Server scanners for example can issue MDS events when they complete the data acquisition from a CAMAC digitizer memory. A plotting definition in the Plot Command Language can then wait for this event before graphing any of the signals on this digitizer.

Note: Since MDS events are used throughout the system care must be taken to pick names which are likely to be unique and not interfere with other users. One method is to prefix your initials to the event name (i.e. TWF_READY_TO_PLOT).

**CONDITION VALUE RETURNED**

MDS$NORMAL The event was successfully issued.
Run-Time Library Routines
MDSS$EVENT

EXAMPLES

This example will issue the MDS event TWF_READY_TO_PLOT without sending any data with the event:

Character*25 EVENT
Data EVENT /'TWF_READY_TO_PLOT'/
Integer*4 STATUS
Integer*4 MDSS$EVENT
.
.
.
STATUS = MDSS$EVENT( EVENT )

This example will issue the same MDS event and pass data with it:

Integer*4 BUFFER(3)
Data BUFFER /100,200,300/
.
.
.
CALL MDSS$EVENT( 'TWF_READY_TO_PLOT' , BUFFER)
MDS$EVENTAST

Enable an MDS event AST.

**FORMAT**

```
MDS$EVENTAST event-name, astadr [,astprm] [,event-id]
```

**RETURNS**

- **VMS Usage:** cond_value
- **Type:** longword (unsigned)
- **Access:** write only
- **Mechanism:** by value

**ARGUMENTS**

- **event-name**
  - **VMS Usage:** mds_event
  - **Type:** character string
  - **Access:** read only
  - **Mechanism:** by descriptor
  - The name of the MDS event.

- **astadr**
  - **VMS Usage:** ast_procedure
  - **Type:** procedure entry mask
  - **Access:** call without stack unwinding
  - **Mechanism:** by reference
  - AST service routine to be executed when the event occurs. The astadr argument is the address of a longword value that is the entry mask to the AST routine.

- **astprm**
  - **VMS Usage:** user_arg
  - **Type:** longword (unsigned)
  - **Access:** read only
  - **Mechanism:** by value
  - An optional value to be passed to the AST routine. The astprm argument is a longword value containing the AST parameter.

- **event-id**
  - **VMS Usage:** event_id
  - **Type:** longword (unsigned)
  - **Access:** write only
  - **Mechanism:** by reference
  - Optionally, the address of a longword to receive an event identification number. This identification number is required if you want to subsequently cancel the AST.
**Run-Time Library Routines**

**MDS$EVENTAST**

**DESCRIPTION**

MDS$EVENTAST requests that the MDS system queue an AST for the requesting process when the specified MDS event is issued. These ASTs are repeating ASTs, that is, once the MDS$EVENTAST routine is invoked, the AST will be executed each time the specified event occurs. It is unnecessary to call MDS$EVENTAST again after each occurrence of the AST. The AST can be canceled by calling the MDS$EVENTCAN procedure.

**Note:** In FORTRAN the AST routine name must be declared using the EXTERNAL statement. If it isn't you will not get a bad status return from MDS$EVENT but your program will probably get an ACCVIO error when the event finally occurs.

<table>
<thead>
<tr>
<th>CONDITION</th>
<th>VALUES</th>
<th>RETURNED</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SS$$_NORMAL</strong></td>
<td></td>
<td>The AST has been successfully enabled.</td>
</tr>
<tr>
<td><strong>SS$$_ACCVIO</strong></td>
<td></td>
<td>The event name cannot be read by the caller.</td>
</tr>
<tr>
<td><strong>SS$$_EXENQLM</strong></td>
<td></td>
<td>The process has exceeded its enqueue limit quota.</td>
</tr>
<tr>
<td><strong>SS$$_IVBUFLEN</strong></td>
<td></td>
<td>The length of the event name was greater than 25 characters.</td>
</tr>
<tr>
<td><strong>LIB$$_INSEF</strong></td>
<td></td>
<td>Insufficient event flags. There were no more event flags available for allocation. The AST cannot be enabled.</td>
</tr>
<tr>
<td><strong>LIB$$_INSVIRMEM</strong></td>
<td></td>
<td>Insufficient virtual memory. The request required more dynamic memory than was available from the operating system.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

This example will enable the routine MY_AST to be queue whenever the event, TWF ready to PLOT, occurs:

```fortran
External MY_AST
.
.
.
CALL MDS$EVENTAST('TWF ready to PLOT', MY_AST)
```
MDSEVENTCAN

Cancel an MDS Event AST

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>MDSEVENTCAN event-id</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETURNS</td>
<td>VMS Usage: cond_value, longword (unsigned), write only, by value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARGUMENTS</th>
<th>event-id</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMS Usage:</td>
<td>event_id, longword (unsigned), read only, by value</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The event identification number returned by a call to MDSEVENTAST.

| DESCRIPTION       | The MDSEVENTCAN routine cancels the AST established by a call to MDSEVENTAST. The AST specified in the MDSEVENTAST call will continue to be executed each time the corresponding event occurs until it is canceled by MDSEVENTAST. |

<table>
<thead>
<tr>
<th>CONDITION VALUES</th>
<th>SS$_NORMAL</th>
<th>The AST has been successfully canceled.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SS$_ACCVIO</td>
<td>The event id was invalid.</td>
</tr>
<tr>
<td></td>
<td>SS$_IVLOCKID</td>
<td>The event id was invalid or already dequeued.</td>
</tr>
</tbody>
</table>
EXAMPLE

The following example shows an event AST being established and later canceled:

```fortran
External MY_AST
Integer*4 EVENT_ID

CALL MDS$EVENTAST('TWF_READY_TO_PLOT', MY_AST,,EVENT_ID) ! Establish event AST

CALL MDS$EVENTCAN(%VAL(EVENT_ID)) ! Cancel the event AST
```
MDS$GETEVI

Get MDS Event information.

**FORMAT**

```
MDS$GETEVI event-id, data-buffer, process-id
```

**RETURNS**

VMS Usage: `cond_value`
type: longword (unsigned)
access: write only
mechanism: by value

**ARGUMENTS**

**event-id**
VMS Usage: `event_id`
type: longword (unsigned)
access: read only
mechanism: by value
The event identification number returned by an MDS$EVENTAST call.

**data-buffer**
VMS Usage: `vector_byte_unsigned`
type: byte (unsigned)
access: write only
mechanism: by reference
The address of a twelve byte buffer to receive the event data loaded when the event was issued.

**process-id**
VMS Usage: `process_id`
type: longword (unsigned)
access: write only
mechanism: by reference
A longword to receive the process identification number of the process which issued the MDS event.

**DESCRIPTION**

The MDS$GETEVI routine returns information about a particular MDS event. The event is accessed via its event identification number. This number is returned by an MDS$EVENTAST call. The information returned is the twelve bytes of data that was optionally supplied when the MDS$EVENT call was issued triggering the event and the PID of the process that issued the event. This information is automatically returned with the MDS$WTEVENT call.

**CONDITION VALUES RETURNED**

MDSRTL 7-39
Run-Time Library Routines

**MDSS$GETEVI**

**SS$NORMAL**  The information was successfully returned.

**SS$ACCVIO**  An routine was unable to write the data or process identification to the addresses provided in the call or an invalid event-id was provided.

**SS$IVLOCKID**  An invalid event-id was provided.

**EXAMPLE**

This example will get the information of the TWF\_READY\_TO\_PLOT event. The GET\_INFO AST routine will be queued whenever the event occurs. In this case the address of the event-id is passed as the argument to the AST. The following program will print out the contents of the data buffer and the PID of the process issuing the event every time the event occurs:

```fortran
External GET\_INFO
Integer*4 EVENT\_ID
CALL MDSS$EVENTAST('TWF\_READY\_TO\_PLOT',GET\_INFO,EVENT\_ID,EVENT\_ID)
CALL SYS$HIBER
END

Subroutine GET\_INFO(EVENT\_ID)
Integer*4 EVENT\_ID
Byte BUFFER(12)
Integer*4 PID
CALL MDSS\$GETEVI(%VAL(EVENT\_ID),BUFFER,PID)
WRITE (*,*) BUFFER,PID
RETURN
END
```

7-40  MDSRTL
MDS$ISTHERE

Check for the presence of an optional argument.

**FORMAT**

MDS$ISTHERE argnum

**RETURNS**

VMS Usage: cond_value
  type: longword (unsigned)
  access: write only
  mechanism: by value

**ARGUMENTS**

argnum
  VMS Usage: longword unsigned
  type: longword (unsigned)
  access: read only
  mechanism: by reference

The number of the argument to the routine. If argnum was equal to one, MDS$ISTHERE would return .TRUE. if the first argument was present and .FALSE. if it was not.

**DESCRIPTION**

MDS$ISTHERE checks for the presence of arguments in the argument frame of a called procedure. Routines with optional arguments can be implemented using MDS$ISTHERE to determine if the argument was supplied or defaulted. Trailing commas are not necessary in the call statement for specifying optional arguments.

Note: Due to the strange way that the FORTRAN compiler and the VM-S Linker handle character string arguments, it is difficult to have FORTRAN procedures with optional character string procedures. There are ways to circumvent this limitation but they are non-trivial and not recommended here.

**CONDITION VALUES RETURNED**

.TRUE. The argument was supplied in the call to this routine.

.FALSE. The argument was not supplied in the call to this routine.
Run-Time Library Routines
MDSS$ISTHERE

EXAMPLE

In the following example, I,K become .TRUE. and J,L become .FALSE.: 

CALL MY_SUB(A,B,C,D)
END
SUBROUTINE MY_SUB(A,B,C,D)
Logical*4 MDS$ISTHERE
Logical*4 I,J,K,L
I = MDS$ISTHERE(1)
J = MDS$ISTHERE(2)
K = MDS$ISTHERE(3)
L = MDS$ISTHERE(4)
RETURN
END
MDS$Q_EVENT

Establish a detection queue for an MDS Event

**FORMAT**

```
MDS$Q_EVENT event
```

**RETURNS**

<table>
<thead>
<tr>
<th>VMS Usage</th>
<th>type</th>
<th>access</th>
<th>mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond_value</td>
<td>longword (unsigned)</td>
<td>write only</td>
<td>by value</td>
</tr>
</tbody>
</table>

**ARGUMENTS**

- **event**
  - VMS Usage: mds_event
  - type: character string
  - access: read only
  - mechanism: by reference
  - The name of an MDS event.

**DESCRIPTION**

MDS$Q_EVENT establishes interest in an MDS event. This routine is used in cases where a program wishes to wait for an event but cannot guarantee that it will be waiting when the event occurs. Using MDS$Q_EVENT, a WFLAND (Wait for And of Events) could be implemented. MDS$Q_EVENT each of the events and then MDS$WTEVENT each of them.

**CONDITION VALUE RETURNED**

<table>
<thead>
<tr>
<th>MDS$NORMAL</th>
<th>MDS$NOEVENT</th>
<th>LIB$INSEF</th>
<th>SSS$ILLEFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>The event was successfully Qed.</td>
<td>The maximum number of Events that can be Qed has been exceeded. Event not Qed.</td>
<td>Insufficient event flags. There were no more event flags available for allocation.</td>
<td>Illegal event flag number specified. Bug/feature of MDS$Q_EVENT</td>
</tr>
</tbody>
</table>
Run-Time Library Routines
MDSSQ_EVENT

EXAMPLE

This example waits for the AND of 4 Events.

```
Integer*4 STATUS
Integer*4 MDSSQ_EVENT
Integer*4 MDSSQ_WTEVENT
.
.
.
CALL MDSSQ_EVENT('EVENT_1')
CALL MDSSQ_EVENT('EVENT_2')
CALL MDSSQ_EVENT('EVENT_3')
CALL MDSSQ_EVENT('EVENT_4')
.
.
.
CALL MDSSQ_WTEVENT('EVENT_1')
CALL MDSSQ_WTEVENT('EVENT_2')
CALL MDSSQ_WTEVENT('EVENT_3')
CALL MDSSQ_WTEVENT('EVENT_4')
.
.
.
```
MDSSFREE1_DD

Free a dynamic block of memory.

**FORMAT**  
MDSSFREE1_DD buffer

**RETURNS**  
VMS Usage: **cond_value**  
type: longword (unsigned)  
access: write only  
mechanism: by value

**ARGUMENTS**  
buffer  
virtual_memory longword (unsigned) modify by descriptor  
A quadword buffer to hold the descriptor of the memory returned. The descriptor has the following format:

```
+---------------+---------------+  
|              31 |              0 |  
+---------------+---------------+  
|          LENGTH |             0 |  
+---------------+---------------+  
|            POINTER |            4 |  
+---------------+---------------+  
```

**DESCRIPTION**  
MDSSFREE1_DD frees a block of virtual memory allocated by a call to MDSSGET1_DD.  
The first longword of the descriptor is a length. Unlike VMS dynamic strings which are limited to 64k bytes, there is no type and class information in the descriptor.

**CONDITION VALUES RETURNED**  
SS$NORMAL  
The memory was successfully returned  
LIB$BADBLOADR  
Pointer field of the record contained an address outside of the space allocated by a call to the MDSSGET1_DD routine.
EXAMPLE

The following example gets a dynamic buffer of 128K bytes and passes it to a subroutine as a real*4 array and then frees it.

Structure DYNAMIC_BUFFER
   Integer*4 LENGTH  ! Length field
   Integer*4 POINTER  ! Pointer Field
END Structure

Record /DYNAMIC_BUFFER/ BUFFER
   Integer*4 MDS$SGET1_DD
   Integer*4 STATUS
.
.
STATUS = MDS$SGET1_DD(131072, BUFFER)  ! Get a buffer (128 K)
IF (STATUS) Then  ! If successful then
   CALL SUB(131072/4, %Val(BUFFER.POINTER))  ! Call the subroutine
.
.
CALL MDS$SFREE1_DD( BUFFER )  ! Free the buffer
END

Subroutine SUB(length,rdata)
   Integer*4 length
   Real*4 rdata(length)
   ! Do something with the data

Return
End
MDS$SGET1_DD

Get a block of virtual memory.

**FORMAT**  
MDS$SGET1_DD size, descriptor

**RETURNS**  
VMS Usage: cond_value
  type: longword (unsigned)
  access: write only
  mechanism: by value

**ARGUMENTS**  
size
  VMS Usage: longword (unsigned)
  type: read only
  access: by reference
  The number of bytes of memory to get.

descriptor
  virtual_memory longword (unsigned) modify by descriptor
  The address of a buffer to receive a virtual memory descriptor as follows:

      31  0
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LENGTH</td>
<td>0</td>
</tr>
<tr>
<td>POINTER</td>
<td>4</td>
</tr>
</tbody>
</table>

**DESCRIPTION**  
MDS$SGET1_DD returns a block of virtual memory of any size. (not restricted to 64K bytes).

The first longword of the descriptor is a length. Unlike VMS dynamic strings which are limited to 64k bytes, There is no type and class information in this descriptor.

**CONDITION VALUES RETURNED**  
| SS$ _NORMAL   | The memory was successfully returned |
| LIB$ INSVIRMEM | Insufficient virtual memory. The request required more dynamic memory than was available from the operating system. No partial assignment (Allocation) is made in this case. |
| LIB$ BADBLOSIZ | Bad block size. The size requested was less than or equal to zero. |

MDSRTL 7-47
EXAMPLE

The following example gets a dynamic buffer of 128K bytes and passes it to a subroutine as a real*4 array.

```
Structure DYNAMIC_BUFFER
  Integer*4 LENGTH      ! Length field
  Integer*4 POINTER    ! Pointer Field
END Structure

Record /DYNAMIC_BUFFER/ BUFFER

STATUS = MDS$GET1_DD(131072, BUFFER) ! Get a buffer (128 K)
IF (STATUS) Then
  CALL SUB(131072/4, %Val(BUFFER.POINTER)) ! Call the subroutine
END

Subroutine SUB(length, rdata)
  Integer*4 length
  Real*4     rdata(length)
  ! Do something with the data
  Return
End
```
MDS$WTEVENT

Wait for an MDS Event

**FORMAT**

MDS$WTEVENT event-name [,data-buffer] [,process-id]

**RETURNS**

<table>
<thead>
<tr>
<th>VMS Usage</th>
<th>cond_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>longword (unsigned)</td>
</tr>
<tr>
<td>access</td>
<td>write only</td>
</tr>
<tr>
<td>mechanism</td>
<td>by value</td>
</tr>
</tbody>
</table>

**ARGUMENTS**

**event-name**

VMS Usage: mds_event

<table>
<thead>
<tr>
<th>type</th>
<th>character string</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism</td>
<td>by descriptor</td>
</tr>
</tbody>
</table>

The name of the MDS event.

**data-buffer**

VMS Usage: vector_byte_unsigned

<table>
<thead>
<tr>
<th>type</th>
<th>byte (unsigned)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>write only</td>
</tr>
<tr>
<td>mechanism</td>
<td>by reference</td>
</tr>
</tbody>
</table>

Optionally, the address of a twelve byte buffer to receive the event data loaded when the event was issued.

**process-id**

VMS Usage: process_id

<table>
<thead>
<tr>
<th>type</th>
<th>longword (unsigned)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>write only</td>
</tr>
<tr>
<td>mechanism</td>
<td>by reference</td>
</tr>
</tbody>
</table>

Optionally, a longword to receive the process identification number of the process which issued the MDS event.

**DESCRIPTION**

The MDS$WTEVENT routine will cause the calling image to be placed in wait state until the specified MDS event takes place. It will optionally return the event information to the caller.

**CONDITION VALUE RETURNED**
Run-Time Library Routines
MDSS$WTEVENT

SS$NORMAL  The event occurred.

EXAMPLE

This example will wait for the TWF_READY_TO_PLOT event and get the process identification of the process issuing the event:

```
Integer*4 PID
CALL MDSS$WTEVENT('TWF_READY_TO_PLOT', PID)
WRITE (*,*) PID
END
```
PCL$DO_COMMAND

Invoke the MDS command line interpreter to execute a PCL command.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>PCL$DO_COMMAND command</th>
</tr>
</thead>
</table>

| RETURNS | VMS Usage: cond_value type: longword (unsigned) access: write only mechanism: by value |

<table>
<thead>
<tr>
<th>ARGUMENTS</th>
<th>command</th>
</tr>
</thead>
</table>

VMS Usage: char_string type: character string access: read only mechanism: by descriptor

A character string command for PCL to execute. (See the Plot Command Language Reference Manual for description of valid PCL commands.)

DESCRIPTION

Using the PCL$DO_COMMAND routine, it is possible to execute any of the commands available in the PCL utility from an application program.

This routine is contained in the shared image, SYS$LIBRARY:MDSDCLSHR.EXE, When linking your program, you must either reference this shared image in an options file or link against a shared image library containing this image.

Note: The commands for manipulating MDS databases such as DIREC-TORY, SET DATABASE, SET DEFAULT, and SET SHOT, are not included in the callable PCL. To use these commands, you must either use the TDB$DO_COMMAND routine or issue the PCL command, SET COMMAND TDB$COMMANDS, to add these database commands to the command list. Likewise, the MDS event commands will not be available unless TDB$COMMANDS is invoked.

<table>
<thead>
<tr>
<th>CONDITION VALUES RETURNED</th>
<th>SS$_NORMAL</th>
<th>The command was successfully executed.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MDSDCL$_UNDEFSYM</td>
<td>The command issued was an unknown command or the syntax of the command was invalid.</td>
</tr>
</tbody>
</table>
Run-Time Library Routines
PCL$DO_COMMAND

**PCL$xxxxxx**

Any error returned from the PCL interface indicating improper sequence of commands or invalid values supplied for qualifiers.

**EXAMPLE**

The following example will plot a MDS database item:

```fortran
CALL PCL$DO_COMMAND('SET COMMAND TDB$COMMANDS') ! Include database commands
CALL PCL$DO_COMMAND('SET DATABASE MY_DATABASE') ! Select database
CALL PCL$DO_COMMAND('DEVICE TT;/DEVNAM=REGIS') ! Select device
CALL PCL$DO_COMMAND('PLOT SH_TOK4_TM SH_TOK4_08') ! Plot data versus time
```
**TDB$ASCTIM**

Converts date field of TDB descriptor to ascii date and time.

**FORMAT**

TDB$ASCTIM date_field, date_time

**RETURNS**

VMS Usage: cond_value
type: longword (unsigned)
access: write only
mechanism: by value

**ARGUMENTS**

date_field
VMS Usage: longword_unsigned
type: longword (unsigned)
access: read only
mechanism: by reference
The longword date field of a TDB descriptor.

date_time
VMS Usage: time_name
type: character string
access: write only
mechanism: by descriptor
The address of a character string descriptor which is to receive the ascii date and time. Since the time is stored in a longword the accuracy of the time is around one second. A string of 20 characters will receive the date string and the time string as follows: dd-mmm-yyyy hh:mmm:ss .

**DESCRIPTION**

The TDB$ASCTIM routine will convert the longword date field of a TDB descriptor to an ascii string of date and time. This longword date field in a TDB descriptor generally refers to the time at which the record was written to the database.

**CONDITION VALUES RETURNED**

SS$_NORMAL Service successfully completed.

SS$_IVTIME Invalid date field in TDB descriptor.
Run-Time Library Routines

TDB$ASCTIM

EXAMPLE

The following is a sample retrieval on the signal "TF_CURRENT":

```
Include 'SYS$LIBRARY:FORMDEF.TLB($TDBDEF)'
! Get TDB descriptor structure
Record /TDB_DESCR/ TDB
! Define descriptor record
Character DATE_STORED*20
! Character string for date/time
CALL TDB$SET_DB('ALCDATA$')
! Set database to "ALCDATA$"
CALL TDB$DATA_D('TF_CURRENT',,,,,,TDB)
! Get the record of TF_CURRENT
! for the current date and shot
CALL TDB$ASCTIM(TDB.DATE,DATE_STORED)
! Convert the date field to ascii
END
! End
```

```
TDB$DATA

Retrieves an MDS database item as a floating point vector.

FORMAT

TDB$DATA name, [date], [shot], [start], [end], [inc], buflen, buffer [,samps-ret [,units [,source [,act-date [,act-shot]]]]]]

RETURNS

VMS Usage: cond_value
  type: longword (unsigned)
  access: write only
  mechanism: by value

ARGUMENTS

name
  VMS Usage: mds_db_item_name
  type: character string
  access: read only
  mechanism: by descriptor

The name of the database item desired. Database access is case sensitive. The name you supply must match exactly the name in the database for retrieval to take place. In almost all cases the database item name will contain only upper case letters.

The name may be prefixed by the database level specification, (for example: “S.MYITEM”, where “S” is the level prefix). If a prefix is included, MDS will search only the levels in the database which have matching prefixes. Otherwise, all levels will be searched for the database item. (See the “Databases” section in the MDS Concepts Manual for details on database levels.)

date
  VMS Usage: date_name
  type: character string
  access: read only
  mechanism: by descriptor

Optionally, the date of the shot desired. If omitted, the date defaults to the date set by TDB$SET_DEFAULT call or the current date if no call to TDB$SET_DEFAULT has been made. The format of the date is dd-mmm-yyyy, or the keyword “TODAY” for the current date, or the keyword “LAST” for the date of the last shot taken.
Run-Time Library Routines
TDB$DATA

shot
VMS Usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference
Optionally, the address of a longword containing the shot number desired. If
omitted, the shot defaults to the shot set by a TDB$SET_DEFAULT call or
the current shot if no call to TDB$SET_DEFAULT has been made. A value
of -1 will use the current shot number.

start
VMS Usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference
Optionally, the address of a longword containing the index of the first sample
in the record you wish to receive. If this argument is omitted or set to 0,
retrieval will begin with the first sample in the record.

end
VMS Usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference
Optionally, the address of a longword containing the index of the last sample
in the record you wish to receive. If this argument is omitted or set to 0,
retrieval will end with the last sample in the record.

inc
VMS Usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference
Optionally, the address of a longword containing the increment between sam-
ple s in the record you wish to receive. For example setting start to 1, end to
100, and inc to 10 would retrieve samples 1, 11, 21, 31, 41, 51, 61, 71, 81, and 91.
If this argument is omitted or set to 0 an increment of 1 will be used.

buflen
VMS Usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference
The address of a longword containing the length in real*4’s of the buffer into
which the data is to be returned.
Run-Time Library Routines
TDB$DATA

buffer
VMS Usage: floating_point
type: F_floating
access: write only
mechanism: by reference
The address of a real*4 variable or array to receive the data. The buffer must
be dimensioned to at least buflen.

samps-ret
VMS Usage: longword_signed
type: longword (signed)
access: write only
mechanism: by reference
Optionally, the address of a longword to receive the number of samples re-
turned in the buffer.

units
VMS Usage: mds_units
type: character string
access: write only
mechanism: by descriptor
An optional string descriptor to receive the units of the data. Units can be
up to 8 characters long.

source
VMS Usage: mds_source
type: character string
access: write only
mechanism: by descriptor
An optional string descriptor to receive the source of the data. The source
of a TDB transform is the username of the user that created the transform.
The source of a digitizer data record is the CSV module name which created
the record.

act-date
VMS Usage: date_name
type: character string
access: write only
mechanism: by descriptor
An optional string descriptor to receive the actual date of the TDB record. For
shot data this date will be the same as that requested. For TDB transforms
this date represents the first day that the transform applies.
### Run-Time Library Routines

#### TDB$DATA

**act-shot**

VMS Usage: longword unsigned

type: longword (unsigned)

access: write only

mechanism: by reference

Optionally, the address of a longword to receive the actual shot number of the TDB record. For shot data this shot number will be the same as that requested. For TDB transforms this shot number represents the first shot that occurred on act-date that the transform applies.

### DESCRIPTION

TDB$DATA will return a specified data record in real+4 format. You must make a call to TDB$SET_DB to establish the current database prior to calling TDB$DATA. If the name supplied is the name of a TDB transform the transformation will be performed as declared. (See the *MDS Command Language Interpreters Reference Manual* for description of transform declarations.)

The error handling for the TDB$DATA call can be specified by using the TDB$SET_ERRORS routine. Errors can be returned as status only, a brief output message, a complete error traceback, or signaled to a condition handler.

### CONDITION VALUES RETURNED

<table>
<thead>
<tr>
<th>TDB$NORMAL</th>
<th>The data has been successfully returned.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDB$TRUNCATED</td>
<td>Part of the data has been successfully returned but the buffer length was not large enough to receive all of the data. Informational.</td>
</tr>
<tr>
<td>TDB$DATERR</td>
<td>An error was encountered when attempting to access or transform the data of the desired item.</td>
</tr>
<tr>
<td>TDB$INVDB</td>
<td>No database was successfully selected using a TDB$SET_DB call prior to this TDB$DATA call.</td>
</tr>
<tr>
<td>SS$IVTIME</td>
<td>An invalid date was specified.</td>
</tr>
</tbody>
</table>

7-58 MDSRTL
EXAMPLES

The following is an sample retrieval on the signal "TF_CURRENT":

```fortran
Real*4 CURRENT(8192)  ! Declare the CURRENT array
CALL TDB$SET_DB('ALCDATA$')  ! Set database to "ALCATOR$
CALL TDB$DATA('TF_CURRENT', 8192, CURRENT)  ! Get the trace of TF_CURRENT
                                  ! for the current date and shot
END  ! End
```

The following example will retrieve every tenth sample of "TF_CURRENT" for shot 99 which occurred on 24th of July 1984:

```fortran
Real*4 CURRENT(819)  ! Declare the CURRENT array
Character*8 UNITS  ! Declare units descriptor
Character*23 SOURCE  ! Declare source descriptor
Integer*4 SAMPLES  ! Declare samples longword
CALL TDB$SET_DB('ALCDATA$')  ! Set database
CALL TDB$DATA('TF_CURRENT', '24-JUL-1984', 99, CURRENT, SAMPLES, UNITS, SOURCE)  ! Get data
                                  + 99,1,10,819, CURRENT, SAMPLES, UNITS, SOURCE
END  ! End
```
TDB$DATA_D

Retrieve a MDS database item in floating point format returned in a TDB descriptor.

**FORMAT**

TDB$DATA_D name, [date], [shot], [start], [end], [inc], tdb_descriptor

**RETURNS**

VMS Usage: cond_value
type: longword (unsigned)
access: write only
mechanism: by_value

**ARGUMENTS**

**name**

VMS Usage: mds_db_item_name
type: character string
access: read only
mechanism: by descriptor

The name of the database item desired. Database access is case sensitive. The name you supply must match exactly the name in the database for retrieval to take place. In almost all cases the database item name will contain only upper case letters.

The name may be prefixed by the database level specification, (for example: “S.MY_ITEM”, where “S.” is the level prefix). If a prefix is included, MDS will search only the levels in the database which have matching prefixes. Otherwise, all levels will be searched for the database item. (See the “Databases” section in the MDS Concepts Manual for details on database levels.)

**date**

VMS Usage: date_name
type: character string
access: read only
mechanism: by descriptor

Optionally, the date of the shot desired. If omitted, the date defaults to the date set by a TDB$SET_DEFAULT call or the current date, if no call to TDB$SET_DEFAULT has been made. The format of the date is dd-mmm-yyyy or the keyword ”TODAY” for the current date or the keyword ”LAST” for the date of the last shot taken.
Run-Time Library Routines

TDB$DATA_D

**shot**
VMS Usage: *longword_signed*
type: *longword (signed)*
access: *read only*
mechanism: *by reference*
Optionally, the address of a longword containing the shot number desired. The shot number defaults to the shot set by a TDB$SET_DEFAULT call or the current shot, if no call to TDB$SET_DEFAULT has been made. A value of -1 will use the current shot number.

**start**
VMS Usage: *longword_signed*
type: *longword (signed)*
access: *read only*
mechanism: *by reference*
Optionally, the address of a longword containing the index of the first sample in the record you wish to receive. If this argument is omitted or set to 0, retrieval will begin with the first sample in the record.

**end**
VMS Usage: *longword_signed*
type: *longword (signed)*
access: *read only*
mechanism: *by reference*
Optionally, the address of a longword containing the index of the last sample in the record you wish to receive. If this argument is omitted or set to 0, retrieval will end with the last sample in the record.

**inc**
VMS Usage: *longword_signed*
type: *longword (signed)*
access: *read only*
mechanism: *by reference*
Optionally, the address of a longword containing the increment between samples in the record you wish to receive. For example setting start to 1, end to 100, and inc to 10 would retrieve samples 1, 11, 21, 31, 41, 51, 61, 71, 81, and 91. If this argument is omitted or set to 0 and increment of 1 will be used.
Run-Time Library Routines
TDB$DATA_D

tdb_descriptor
VMS Usage: tdb_descriptor
type: F-floating
access: write only
mechanism: by descriptor

The address of a TDB descriptor which will be filled in by this call. The TDB descriptor structure is defined in the $TDBDEF include module in the SYS$LIBRARY:FORMDSDEF.TLB text library. The structure of a TDB descriptor is as follows:

```
+---------+---------+---------+---------+
| 31      | 0       | 4       | 8       |
| LENGTH  | POINTER | SHOTID  |
+---------+---------+---------+---------+
| reserved| FTYPE   | CLASS   | DTYPE   |
+---------+---------+---------+---------+
| SOURCE  |
| (23 bytes) |
+---------+---------+---------+---------+
| UNITS   |
| (8 bytes) |
| DATE-TIME (cont) |
+---------+---------+---------+---------+
```

DESCRIPTION
TDB$DATA_D will return a TDB descriptor pointing to specified data record. You must make a call to TDB$SET_DB to establish the current database prior to calling TDB$DATA_D. If the name supplied is the name of a TDB transform the transformation will be performed as declared. (See the MDS Command Language Interpreters Reference Manual for description of transform declarations.)

The record is stored in dynamic memory. If you re-use the descriptor in subsequent calls to TDB$DATA_D the dynamic memory will be re-used or freed as needed. You can release this dynamic memory manually via a call to TDB$FREE1-DD. Because a TDB descriptor does point to dynamic storage you should not modify the length or pointer fields of the descriptor directly.

Errors generated by calls to TDB$DATA_D can be reported in several ways which are selectable via calls to TDB$SET_ERRORS. Errors can be returned as status only, brief message output, full message output, or signaled to a condition handler.

<table>
<thead>
<tr>
<th>CONDITION</th>
<th>VALUES</th>
<th>RETURNED</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDB$NORMAL</td>
<td></td>
<td>The data has been successfully returned.</td>
</tr>
</tbody>
</table>

7-62 MDSRTL
Run-Time Library Routines

TDB$DATA_D

**TDB$_DATERR**
An error was encountered when attempting to access or transform the data of the desired record.

**TDB$_INVDB**
No database was successfully selected using a TDB$SET_DB call prior to this TDB$DATA call.

**SS$_IVTIME**
An invalid date was specified.

**EXAMPLE**

The following is an sample retrieval on the signal "TF_CURRENT":

```plaintext
Include 'SYS$LIBRARY:FORMD$DEF($TDB$DEF)' ! Declare TDB_DESCR structure
Record /TDB_DESCR/ TDB
CALL TDB$SET_DB('ALCDATA$') ! Declare record for descriptor
CALL TDB$DATA_D('TF_CURRENT', TDB)
CALL SUB(TDB.LENGTH/4,%VAL(TDB.POINTER)) ! Get the trace of TF_CURRENT
CALL TDB$FREE1_DD(TDB) ! for the current date and shot
END
Subroutine SUB(length,rdata)
Integer*4 length
Real*4 rdata(length)

! Call subroutine to access data
! Free the dynamic memory
! End

! Do something with the data
Return
End
```
Run-Time Library Routines
TDB$DATE_SHOT

TDB$DATE_SHOT

Convert encoded shot identification to date and shot.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>TDB$DATE_SHOT [shotid] , [date] , [shot]</th>
</tr>
</thead>
</table>

| RETURNS | VMS Usage: cond_value type: longword (unsigned) access: write only mechanism: by value |

<table>
<thead>
<tr>
<th>ARGUMENTS</th>
<th>shotid</th>
<th>VMS Usage: mds_shotid type: longword (unsigned) access: read only mechanism: by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Optionally, the address of a longword or TDB descriptor shotid field containing a TDB shot identification number. If omitted, the current default date and shot will be returned.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>date</th>
<th>VMS Usage: date_name type: character string access: write only mechanism: by descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Optionally, the address of a character string descriptor to receive the ascii date of the shot. The date will be returned as dd-mmm-yyyy.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>shot</th>
<th>VMS Usage: longword_signed type: longword (signed) access: write only mechanism: by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Optionally, the address of a longword to receive the shot number.</td>
</tr>
</tbody>
</table>

| DESCRIPTION | MDS uses an encoded shot identification number for indexing records in the database call a SHOTID. The TDB$DATE_SHOT routine will convert an encoded SHOTID into an ascii date (dd-mmm-yyyy) and an integer shot number. |

| CONDITION VALUES RETURNED | TDB$NORMAL | The service was successfully completed. |
Run-Time Library Routines

TDB$DATE_SHOT

TDB$INVDB

No MDS database is currently open, set to an MDS database first.

EXAMPLE

The following will convert a shotid to date and shot:

Include 'SYS$LIBRARY:FORMD$DEF($TDB$DEF)'
Record /TDB_DESCR/ TDB_DESC
Character*11 DATE
Integer*4 SHOT
CALL TDB$SET_DB ('MY_DATABASE')
CALL TDB$DATA_D ('MY_SIGNAL', , , , , , TDB_DESC)
CALL TDB$DATE_SHOT (TDB_DESC, SHOTID, DATE, SHOT)
WRITE (*,*) DATE, SHOT
END

! Declare tdb descriptor
! Declare string for date
! Declare integer*4 for shot
! Select a database
! Get signal from database
! Convert shotid to date and shot
! Write out the date and shot
! End
Run-Time Library Routines
TDB$DO_COMMAND

TDB$DO_COMMAND
Invokes the MDS command line interpreter to execute a TDB command (database access and MDS events).

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>TDB$DO_COMMAND command</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>RETURNS</th>
<th>VMS Usage: cond_value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>type: longword (unsigned)</td>
</tr>
<tr>
<td></td>
<td>access: write only</td>
</tr>
<tr>
<td></td>
<td>mechanism: by value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARGUMENTS</th>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMS Usage:</td>
<td>char_string</td>
</tr>
<tr>
<td>type:</td>
<td>character string</td>
</tr>
<tr>
<td>access:</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism:</td>
<td>by descriptor</td>
</tr>
</tbody>
</table>

A character string command to execute. (See the *MDS Command Language Interpreters Reference Manual* for description of valid database commands.)

<table>
<thead>
<tr>
<th>DESCRIPTION</th>
</tr>
</thead>
</table>
Using the TDB$DO_COMMAND routine, it is possible to execute any of the database commands or MDS event commands, available in the several of the MDS command line interpreter utilities, from an application program. Most of these command functions can be executed more efficiently with other runtime library functions.

This routine is contained in the shared image, SYS$LIBRARY:MDSDCLSHR.EXE, When linking your program, you must either reference this shared image in an options file or link against a shared image library containing this image.

<table>
<thead>
<tr>
<th>CONDITION VALUES RETURNED</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>SS$_NORMAL</th>
<th>The command was successfully executed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDSDCL$_UNDEFSYM</td>
<td>The command issued was an unknown command or the syntax of the command was invalid.</td>
</tr>
<tr>
<td>TDB$_xxxxxx</td>
<td>Any error returned from the TDB interface indicating improper sequence of commands or invalid values supplied for qualifiers.</td>
</tr>
</tbody>
</table>
EXAMPLE

The following example will select an MDS database, wait for a shot to occur, and produce a directory of that shot:

CALL TDB$DO_COMMAND('SET DATABASE MY_DATABASE') ! Select database
CALL TDB$DO_COMMAND('WFEVENT SHOT_DONE') ! Wait for an event
CALL TDB$DO_COMMAND('DIRECTORY') ! Get a directory of the shot
TDB$EVENT_SHOT

Generate an MDS EVENT with the shot identification of the current shot of a database as the data for the EVENT.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>TDB$EVENT_SHOT event, [date], [shot]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>RETURNS</th>
<th>VMS Usage: cond_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>type:</td>
<td>longword (unsigned)</td>
</tr>
<tr>
<td>access:</td>
<td>write only</td>
</tr>
<tr>
<td>mechanism:</td>
<td>by value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARGUMENTS</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMS Usage:</td>
<td>mds_event</td>
</tr>
<tr>
<td>type:</td>
<td>character string</td>
</tr>
<tr>
<td>access:</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism:</td>
<td>by descriptor</td>
</tr>
</tbody>
</table>

The name of the MDS event to generate.

<table>
<thead>
<tr>
<th>ARGUMENTS</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMS Usage:</td>
<td>time_name</td>
</tr>
<tr>
<td>type:</td>
<td>character string</td>
</tr>
<tr>
<td>access:</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism:</td>
<td>by descriptor</td>
</tr>
</tbody>
</table>

Optionally the date of the shot to use for the shot identification in the data block of the event.

<table>
<thead>
<tr>
<th>ARGUMENTS</th>
<th>shot</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMS Usage:</td>
<td>mds_shot_number</td>
</tr>
<tr>
<td>type:</td>
<td>Longword Integer (signed)</td>
</tr>
<tr>
<td>access:</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism:</td>
<td>by reference</td>
</tr>
</tbody>
</table>

Optionally the shot number to use for the shot identification in the data block of the event.

| DESCRIPTION | TDB$EVENT_SHOT routine will generate an event and place an MDS shot identification into the data block associated with the event. This routine can be used in conjunction with TDB$GET_SHOT to implement a queue of shots to be processed. |

<table>
<thead>
<tr>
<th>Condition Values Returned</th>
<th>MDS$NORMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>The event was successfully issued.</td>
<td></td>
</tr>
</tbody>
</table>
Run-Time Library Routines

TDB$EVENT_SHOT

SS$_IVTIME

The date supplied is not a valid date format, use dd-mmm-yyyy format.

TDB$_INVDB

There has been no call to TDB$SET_DB to establish the current database. It is necessary to call select a database to determine whether the shot identification number will be encoded in shot only or date/shot format.

EXAMPLE

This example will generate the EVENT MY_ANALYSIS_DONE for the current shot.

Call TDB$SET_DB('MY_DATA') ! set the database
.
.
Call TDB$EVENT_SHOT('MY_ANALYSIS_DONE',)
END
TDB$FILE_NAME

Construct database file name from prefix and shotid.

**FORMAT**

TDB$FILE_NAME prefix, shotid, file-name

**RETURNS**

VMS Usage: cond_value
- type: longword (unsigned)
- access: write only
- mechanism: by value

**ARGUMENTS**

prefix
- VMS Usage: char_string
- type: character string
- access: read only
- mechanism: by descriptor

The database level identifier. A character string of length 1 which identifies the database level as defined in the database definition (DBLEVEL commands).

shotid
- VMS Usage: mds_shotid
- type: longword (unsigned)
- access: read only
- mechanism: by reference

The address of a longword containing an encoded shot identification number.

file-name
- VMS Usage: file_spec
- type: character string
- access: write only
- mechanism: by descriptor

The address of a character string descriptor to receive the ascii file specification.

**DESCRIPTION**

Construct the name of the database file based on the specified database level and shot identification. You must make a call to TDB$SET_DB before calling this routine.

**CONDITION VALUES RETURNED**

- TDB$NORMAL: The file spec has been successfully returned.
- TDB$INVDB: No database was selected using a TDB$SET_DB call prior to this call.
EXAMPLE

The following example will return the database file specification for level "S", shot 99, and date 17-MAR-1987:

```
Integer*4 SHOTID
Character*32 FILNAM

CALL TDB$SET_DB('MY_DATABASE')    ! Select database
CALL TDB$SHOT_ID('17-MAR-1987',99,SHOTID)    ! Get shotid
CALL TDB$FILE_NAME('S',SHOTID,FILNAM)    ! Get file spec
WRITE (*,*),FILNAM                   ! Write out file name
END                                   ! End
```
TDB$FIND_RECORD

Search an MDS shot for a data item and return it’s name and optionally a descriptor of the item. This routine will accept wild-card names and return a sequence of names.

**FORMATT**

TDB$FIND_RECORD name_in, [date], [shot],
context, name_out
[,tdb_descriptor]

**RETURNS**

<table>
<thead>
<tr>
<th>VMS Usage:</th>
<th>cond_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>type:</td>
<td>longword (unsigned)</td>
</tr>
<tr>
<td>access:</td>
<td>write only</td>
</tr>
<tr>
<td>mechanism:</td>
<td>by value</td>
</tr>
</tbody>
</table>

**ARGUMENTS**

**name_in**

VMS Usage: mds_db_item_name

| type:      | character string |
| access:    | read only |
| mechanism: | by descriptor |

The name of the database item desired to find. Standard VMS wild card characters may be present in the name. Database access is case sensitive. In almost all cases the database item name will contain only upper case letters. TDB$FIND_RECORD will return the next record in the shot which matches the wild card specification.

The name may be prefixed by the database level specification, (for example: “S.MY_ITEM”, where “S.” is the level prefix). If a prefix is included, MDS will search only the levels in the database which have matching prefixes. Otherwise, all levels will be searched for the database item. (See the “Databases” section in the MDS Concepts Manual for details on database levels.)

**date**

VMS Usage: date_name

| type:      | character string |
| access:    | read only |
| mechanism: | by descriptor |

Optionally, the date of the shot desired. If omitted, the date defaults to the date set by a TDB$SET_DEFAULT call or the current date, if no call to TDB$SET_DEFAULT has been made. The format of the date is dd-mm-yyyy or the keyword "TODAY” for the current date or the keyword ”LAST” for the date of the last shot taken.
Run-Time Library Routines
TDB$FIND_RECORD

shot
VMS Usage: longword_signed
Type: longword (signed)
Access: read only
Mechanism: by reference
Optionally, the address of a longword containing the shot number desired. The shot number defaults to the shot set by a TDB$SET_DEFAULT call or the current shot, if no call to TDB$SET_DEFAULT has been made. A value of -1 will use the current shot number.

context
VMS Usage: longword_signed
Type: longword (signed)
Access: read write
Mechanism: by reference
This argument is used to tell FIND_RECORD whether or not to begin a new search or return the record in the previous search. It should be set to 0 to begin a new record search. It should not be modified accept to reset it to zero to begin a new search.

name_out
VMS Usage: mds_db_item_name
Type: character string
Access: write only
Mechanism: by descriptor
The address of a character string descriptor to hold the name of the record found. This argument is required even if this is not a wild card lookup.
**tdb_descriptor**

VMS Usage: **tdb_descriptor**

type: **F-float**

access: **write only**

mechanism: **by reference**

Optionally the address of a TDB descriptor which will be filled in by this call with information about the record found. The TDB descriptor structure is defined in the $TDBDEF include module in the SYS$LIBRARY:FORMDSDEF.TLB text library. The structure of a TDB descriptor is as follows:

```
LENGTH 0
POINTER 4
SHOTID 8
reserved FTYPE CLASS DTYPE 12
SOURCEx28 by bytes 29
UNITS 24 28 32 36
DATE-TIME 40 44
DATE-TIME (cont) 48
```

**DESCRIPTION**

TDB$FIND_RECORD will search a shot of an MDS database for a signal specified by a wildcard string. It will return the name(s) of any signals matching the specification. Optionally it will also return a TDB_DESCRIPTOR with information about the record that was found.

**Note:** This TDB_DESCRIPTOR does actually point at any memory. On return from a TDB$FIND_RECORD call the descriptor is filled in (including the length field) with the information from the database item found.

Errors generated by calls to TDB$DATA_D can be reported in several ways which are selectable via calls to TDB$SET_ERRORS. Errors can be returned as status only, brief message output, full message output, or signaled to a condition handler.

<table>
<thead>
<tr>
<th>CONDITION VALUES RETURNED</th>
<th>TDB$_NORMAL</th>
<th>TDB$_NMR</th>
</tr>
</thead>
<tbody>
<tr>
<td>The data has been successfully returned.</td>
<td>No more records matching specification.</td>
<td></td>
</tr>
</tbody>
</table>
Run-Time Library Routines
TDB$FIND_RECORD

TDB$_DATERR  An error was encountered when attempting to access or transform the data of the desired record.

TDB$_INVDB  No database was successfully selected using a TDB$SET_DB call prior to this TDB$DATA call.

SS$_IVTIME  An invalid date was specified.

EXAMPLE

The following is an sample will display the names of all of signals matching "TF_CURRENT_*":

```plaintext
Include 'SYS$LIBRARY:FORMD$DEF($TDBDEF)'
! Declare TDB stuff
Integer*4  CTX
Character*(TDB$$_NAME) NAME
CALL TDB$SET_DB('ALCDATA$')
! Set database
Do While (TDB$FIND_RECORD('TF_CURRENT_*', CTX, NAME))
  Call LIB$PUT_OUTPUT('Found: ' //NAME)
End do
! print it out
End
! End do
```
Run-Time Library Routines

TDB$FINISH

TDB$FINISH

Close all open database files and release all temporary database buffers.

FORMAT

TDB$FINISH

RETURNS

VMS Usage: cond_value
type: longword (unsigned)
access: write only
mechanism: by value

ARGUMENTS

None.

DESCRIPTION

The TDB$FINISH routine will close all TDB files that may be open and release all temporary file buffers associated with them. This operation is performed automatically for you upon image termination so it is normally not necessary to do it explicitly.

CONDITION VALUE RETURNED

TDB$NORMAL The service was successfully completed.
Run-Time Library Routines

TDB$GET

Retrieve MDS database item (unprocessed).

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>TDB$GET name, shotid, tdb-desc</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>RETURNS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VMS Usage:</td>
<td>cond_value</td>
</tr>
<tr>
<td>type:</td>
<td>longword (unsigned)</td>
</tr>
<tr>
<td>access:</td>
<td>write only</td>
</tr>
<tr>
<td>mechanism:</td>
<td>by value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARGUMENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>mds_db_item_name</td>
</tr>
<tr>
<td>type:</td>
<td>character string</td>
</tr>
<tr>
<td>access:</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism:</td>
<td>by descriptor</td>
</tr>
</tbody>
</table>

The name of the database item desired. Database access is case sensitive. The name you supply must match exactly the name in the database for retrieval to take place. In almost all cases the database item name will contain only upper case letters.

The name may be prefixed by the database level specification, (for example: “S.MY_ITEM”, where “S.” is the level prefix). If a prefix is included, MDS will search only the levels in the database which have matching prefixes. Otherwise, all levels will be searched for the database item. (See the “Databases” section in the MDS Concepts Manual for details on database levels.)

shotid | mds_shotid |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VMS Usage:</td>
<td>longword (unsigned)</td>
</tr>
<tr>
<td>type:</td>
<td>read only</td>
</tr>
<tr>
<td>access:</td>
<td>by reference</td>
</tr>
</tbody>
</table>

The address of a longword containing the encoded shot identification number of the desired shot.
Run-Time Library Routines
TDB$GET

**tdb-desc**

VMS Usage: **tdb descriptor**

type: **one of the tdb data types**

access: **modify**

mechanism: **by descriptor**

The address of a TDB descriptor which will be filled in by this call. The TDB descriptor structure is defined for FORTRAN in the $TDBDEF module in the SYS$LIBRARY:FORMDFDEF.TLB text library. This structure is defined as follows:

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENGTH</td>
<td>0</td>
</tr>
<tr>
<td>POINTER</td>
<td>4</td>
</tr>
<tr>
<td>SHOTID</td>
<td>8</td>
</tr>
<tr>
<td>reserved</td>
<td>12</td>
</tr>
<tr>
<td>FTYPE</td>
<td>16</td>
</tr>
<tr>
<td>CLASS</td>
<td>20</td>
</tr>
<tr>
<td>DTYPE</td>
<td>24</td>
</tr>
<tr>
<td>SOURCE</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>32</td>
</tr>
<tr>
<td>UNITS</td>
<td>36</td>
</tr>
<tr>
<td>DATE-TIME</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>44</td>
</tr>
<tr>
<td>DATE-TIME (cont)</td>
<td>48</td>
</tr>
</tbody>
</table>
```

**DESCRIPTION**

This routine will return a TDB descriptor pointing to a database item. No transformation of the data is performed. The record is stored in dynamic memory. If you re-use the descriptor in subsequent calls, the dynamic memory will be re-used or freed as needed. You can release this dynamic memory manually via a call to TDB$SFREE_DD. Because a TDB descriptor does point to dynamic storage you should not modify the length of pointer fields of the descriptor directly.

All errors occurring during a TDB$GET call will be signaled. Calls to TDB$SET_ERRORS have no effect on the error handling of this routine.

<table>
<thead>
<tr>
<th>CONDITION VALUE</th>
<th>TDB$NORMAL</th>
<th>TDB$RNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETURNED</td>
<td>The data has been successfully returned.</td>
<td>Error retrieving MDS database item. The signaled error will contain further information describing the reason for the failure.</td>
</tr>
</tbody>
</table>
EXAMPLE

The following example will retrieve a record from the MDS database:

Include 'SYS$LIBRARY:FORMD$SDEF($TDBDEF)'

Record /TDB_DESCR/ TDB_D

CALL TDB$SET_DB('MY_DATABASE')

CALL TDB$SHOT_ID('',SHOTID)

CALL TDB$GET('S.NY_SIGNAL',SHOTID,TDB_D)

WRITE (*,*) TDB_D.LENGTH,TDB_D.SOURCE

END
TDB$GETDBI

Get MDS database information

FORMAT

TDB$SET_DB itmLst

RETURNS

VMS Usage: cond_value
  type: longword (unsigned)
  access: write only
  mechanism: by value

ARGUMENTS
**Run-Time Library Routines**

**TDB$GETDBI**

**itm1st**

VMS Usage: **item_list_3**

type: **longword (unsigned)**

access: **read only**

mechanism: **by reference**

Item list specifying which information about the MDS database is to be returned. The `itm1st` argument is the address of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by a longword of 0. The following diagram depicts a single item descriptor.

```
+----------+----------+
| item code| buffer length |
+----------+----------+
      |
|       |          |
| buffer address |
+----------+----------+
      |
| return length address |
+-----------------------+
```

**TDB$GETDBI Item Descriptor Fields**

**buffer length**

A word containing a user-supplied integer specifying the length (in bytes) of the buffer in which TDB$GETDBI is to write the information. The length of the buffer needed depends upon the `item code` specified in the `item code` field of the item descriptor. If the value of `buffer length` is too small, TDB$GETDBI truncates the data.

**item code**

A word containing a user-supplied symbolic code specifying the item of information that TDB$GETDBI is to return. These codes are defined by the $DBIDEF include module in the SYS$LIBRARY:FORMD$DEF.TLB text library. A description of each `item code` is given in the “TDB$GETDBI Item Codes” section.

**buffer address**

A longword containing the user-supplied address of the buffer in which TDB$GETDBI writes the length in bytes of the information it actually returned.

**return length address**

A longword containing the user-supplied address of a longword in which TDB$GETDBI writes the length in bytes of the information it actually returned. The return length is only updated for `item code` values which return character strings.

**TDB$GETDBI Item Codes**

**DBI$NAME**

When `DBI$NAME` is specified, TDB$GETDBI returns the database name as defined in the database definition file.

**DBI$SHOPTID_FORMAT**

When `DBI$SHOPTID_FORMAT` is specified, TDB$GETDBI returns a value
Run-Time Library Routines
TDB$GETDBI

of 1 if a SHOT_ONLY database, otherwise it returns a value of 0 to the user buffer.

DBI$_SHOTID_FILE
When DBI$_SHOTID_FILE is specified, TDB$GETDBI returns the name of the file which contains the current shot identification.

DBI$_FULL_NAME
When DBI$_FULL_NAME is specified, TDB$GETDBI returns the full file specification of the database definition file.

DBI$_PREFIXES
When DBI$_PREFIXES is specified, TDB$GETDBI returns a string of the available prefixes in the database.

DBI$_MAX_FILES
When DBI$_MAX_FILES is specified, TDB$GETDBI returns the value set for the maximum number of database files which are to be concurrently open.

DBI$_LEVEL
When DBI$_LEVEL is specified, TDB$GETDBI uses the value of the long-word whose address is supplied in the buffer address field to determine which level to use for subsequent item codes which refer to database levels (DBI$_MASK, DBI$_DESCRIPTION, DBI$_SHELL, DBI$_TEMPLATE, DBI$_READONLY, and DBI$_PUT_BUFFERS). Levels start at level 1.

DBI$_MASK
When DBI$_MASK is specified, TDB$GETDBI returns the file mask used in constructing file specifications for level 1 in the database or the current level, dictated by a preceding DBI$_LEVEL item descriptor.

DBI$_SHELL
When DBI$_SHELL is specified, TDB$GETDBI returns the file file specification of the shell file to be used for level 1 in the database or the current level, dictated by a preceding DBI$_LEVEL item descriptor.

DBI$_TEMPLATE
When DBI$_TEMPLATE is specified, TDB$GETDBI returns the file specification of the template file to be used for level 1 in the database or the current level, dictated by a preceding DBI$_LEVEL item descriptor.

DBI$_READONLY
When DBI$_READONLY is specified, TDB$GETDBI returns a value of 1 if the current level is readonly or 0 if not.

DBI$_PUT_BUFFERS
When DBI$_PUT_BUFFERS is specified, TDB$GETDBI returns the number of buffers to use for put operations for level 1 in the database or the current level, dictated by a preceding DBI$_LEVEL item descriptor.
Run-Time Library Routines

**TDB$GETDBI**

### DESCRIPTION

The TDB$GETDBI routine will return information about the current MDS database. You must perform a TDB$SET_DB call to establish a database context before calling this routine.

See the “DATABASE” section of the *MDS Concepts Manual* for more detailed information on database definition files.

### CONDITION VALUES RETURNED

<table>
<thead>
<tr>
<th>CONDITION</th>
<th>VALUES</th>
<th>RETURNED</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDB$_NORMAL</td>
<td>The database information has been successfully returned.</td>
<td></td>
</tr>
<tr>
<td>SS$_BADPARAM</td>
<td>An invalid item code was specified or an invalid level was referenced for the database.</td>
<td></td>
</tr>
</tbody>
</table>
EXAMPLE

The following example will obtain the name of the database and the shell file used for the level "S" of the database.

Implicit none
Include 'SYSLIBRARY:FORMDSDEF($DBIDEF)'
Structure ITEM/
  Union
    Map
      Integer*2 BUFFER_LENGTH
      Integer*2 CODE
      Integer*4 BUFFER_ADDRESS
      Integer*4 RETURN_LENGTH_ADDRESS
    End Map
    Map
      Integer*4 END_OF_LIST /0/
    End Map
  End Union
End structure
Record ITEM/ITEMST(5)
Character*32 DATABASE_NAME
Character*255 SHELL
Integer*4 SHELL_LENGTH
Integer*4 DATABASE_LENGTH
Character*32 PREFIXES
Integer*4 PREFIXES_LENGTH
Integer*4 LEVEL
CALL TDB$SET_DB ('MY_DATABASE')
ITEMST(1).BUFFER_LENGTH = 32
ITEMST(1).CODE = DBI$PREFIXES
ITEMST(1).BUFFER_ADDRESS = %LOC(PREFIXES)
ITEMST(1).RETURN_LENGTH_ADDRESS = %LOC(PREFIXES_LENGTH)
ITEMST(2).END_OF_LIST = 0
CALL TDB$GETDBI (ITEMST)
LEVEL = INDEX(PREFIXES(1:PREFIXES_LENGTH), 'S')
ITEMST(1).BUFFER_LENGTH = 32
ITEMST(1).CODE = DBI$NAME
ITEMST(1).BUFFER_ADDRESS = %LOC(DATABASE_NAME)
ITEMST(1).RETURN_LENGTH_ADDRESS = %LOC(DATABASE_LENGTH)
ITEMST(2).BUFFER_LENGTH = 4
ITEMST(2).CODE = DBI$LEVEL
ITEMST(2).BUFFER_ADDRESS = %LOC(LEVEL)
ITEMST(3).BUFFER_LENGTH = 255
ITEMST(3).CODE = DBI$SHELL
ITEMST(3).BUFFER_ADDRESS = %LOC(SHELL)
ITEMST(3).RETURN_LENGTH_ADDRESS = %LOC(SHELL_LENGTH)
ITEMST(4).END_OF_LIST = 0
CALL TDB$GETDBI(ITEMLIST)
WRITE (*,*) 'Database name is : ', DATABASE_NAME(1:DATABASE_LENGTH)
WRITE (*,*) 'Shell file for level S is : ', SHELL(1:SHELL_LENGTH)
END
Run-Time Library Routines
TDB$GET_SHOT

TDB$GET_SHOT

Wait for an MDS event and return the shot identification of the current shot. The events and corresponding shot numbers are queued so shots cannot be missed.

**FORMAT**

TDB$GET_SHOT event-name, date, shot

**RETURNS**

VMS Usage: cond_value
type: longword (unsigned)
access: write only
mechanism: by value

**ARGUMENTS**

**event-name**

VMS Usage: mds_event
type: character string
access: read only
mechanism: by descriptor
The name of the event which is used to indicate a shot has occurred.

**date**

date_name character string write only by descriptor The address of a character string descriptor to contain the date string returned. The date will be returned in the format, dd-mmm-yyyy.

**shot**

longword_unsigned longword (unsigned) write only by descriptor The address of a longword to receive the shot number.

**DESCRIPTION**

The first time TDB$GET_SHOT is called, it constructs a shot identification queue, establishes an MDS event AST routine to catch the specified event and waits for the specified event to occur. Once the AST has been established, each occurrence of the event will cause the shot identification, at the time of the event, to be placed in the queue. Subsequent calls to TDB$GET_SHOT will remove an entry from the queue (first in first out) and return the shot identification to the caller. If the queue is empty, TDB$GET_SHOT will wait for the specified event before returning a shot identification.

**CONDITION VALUE RETURNED**

**SS$_NORMAL**

A DATE and a SHOT were returned.
EXAMPLE

This example will call the subroutine, DANAL, each time the event, MY_DATA_READY, occurs:

```plaintext
Integer*4 TDB$GET_SHOT
Integer*4 SHOT
Character*11 DATE
Do While (TDB$GET_SHOT('MY_DATA_READY', DATE, SHOT))
   CALL DANAL(DATE, SHOT)
END DO
END
```
TDB$PUT

Write an MDS database item in an MDS database.

FORMAT

TDB$PUT name, tdbdesc

RETURNS

VMS Usage: cond_value
  type: longword (unsigned)
  access: write only
  mechanism: by value

ARGUMENTS

name
  VMS Usage: mds_db_item_name
  type: character string
  access: read only
  mechanism: by descriptor

The name of the database item to store. The name MUST be prefixed by the database level in which the item is to be written (for example: “S.MY_ITEM”, where “S.” is the level prefix). (See the “Databases” section in the MDS Concepts Manual for details on database levels.)
Run-Time Library Routines

TDB$PUT

tdbdesc
VMS Usage: *tdb_descriptor*
type: longword (unsigned)
access: read only
mechanism: by descriptor

The address of a TDB descriptor pointing to the data to be written. The shot identification number (SHOTID) in the descriptor is used to determine the shot in which the record is to be stored. The structure for a TDB descriptor is defined, for FORTRAN, in the $TDBDEF module of the SYS$LIBRARY:FORMDSDEF.TLB text library. This structure is defined as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENGTH</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>POINTER</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>SHOTID</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>FTYPE</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>CLASS</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>DTYPE</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>SOURCE</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>UNITS</td>
<td>45</td>
<td>8</td>
</tr>
<tr>
<td>DATE-TIME</td>
<td>53</td>
<td>8</td>
</tr>
</tbody>
</table>

DESCRIPTION
The TDB$PUT service will write a TDB record to the database. This operation is done asynchronously. The record is copied to a temporary buffer so the memory pointed to by the TDB descriptor may be reused immediately after control returns to the calling procedure. The name of the record must be prefixed by the file type the record is to be stored in. If the file does not exist it is automatically created, assuming the caller has sufficient privileges to create a file in the database root.

All errors are signaled. Calls to TDB$SET_ERRORS will not affect the error handling of this routine.

<table>
<thead>
<tr>
<th>CONDITION VALUES RETURNED</th>
<th>RMS$_NORMAL</th>
<th>RMS$_PENDING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Synchronous operation completed. Success</td>
<td>Asynchronous operation not yet completed. Success</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONDITION VALUE SIGNALED</th>
<th>RMS$_NORMAL</th>
<th>RMS$_PENDING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Synchronous operation completed. Success</td>
<td>Asynchronous operation not yet completed. Success</td>
</tr>
</tbody>
</table>
Run-Time Library Routines
TDB$PUT

TDB$_PUTERR Error writing the database item. The signaled error will contain more information as to the cause of the failure.

EXAMPLE

The following example will put a 12-bit data record into the MDS database:

```plaintext
OPTIONS /CHECK=NOOVERFLOW
Include 'SYS$LIBRARY:FORMX$DEF($TDBDEF)'
Record /TDB_DESCR/ TDBDESC
External TDB$K_DTYPE_12BITU
External TDB$K_CLASS_S
Integer*2 BUFFER (4096)
CALL TDB$SET_ID('MYDATABASE')
CALL GET_DATA (BUFFER)
TDBDESCR.LENGTH = 8192
TDBDESCR.POINTER = %LOC (BUFFER)
TDBDESCR.DTYPE = %LOC (TDB$K_DTYPE_12BITU)
TDBDESCR.CLASS = %LOC (TDB$K_CLASS_S)
TDBDESCR.SOURCE = 'MY_PROGRAM'
TDBDESCR.UNITS = 'counts'
CALL TDB$SHOT_ID(,,TDBDESCR.SHOTID)
CALL TDB$PUT('S_MY_ITEM',TDBDESCR)
END
```

No overflow for dtype
Get TDB descriptor structure
Declare TDB descriptor
Datatype (from TDB$DTYPE$S.EXE/SHARE)
Class (from TDB$DTYPE$S.EXE/SHARE)
Buffer containing data
Select database
Get some data (user routine)
Fill in the descriptor
Get shot id
Put the data
End
**TDB$PUT_DATA**

Write floating point database item to MDS database.

**FORMAT**  

```
TDB$PUT_DATA name, [date], [shot], buflen, buffer [,units])
```

**RETURNS**

VMS Usage: `cond_value`  

<table>
<thead>
<tr>
<th>type:</th>
<th>longword (unsigned)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access:</td>
<td>write only</td>
</tr>
<tr>
<td>mechanism:</td>
<td>by value</td>
</tr>
</tbody>
</table>

**ARGUMENTS**

**name**

VMS Usage: `mds_db_item_name`  

<table>
<thead>
<tr>
<th>type:</th>
<th>character string</th>
</tr>
</thead>
<tbody>
<tr>
<td>access:</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism:</td>
<td>by descriptor</td>
</tr>
</tbody>
</table>

The name of the database item to be created. The name may be prefixed by the database level specification, (for example: “S.MY_ITEM”, where “S.” is the level prefix). If a prefix is included, MDS will place the item in the first level in the database which has a matching prefix. Otherwise, the database item will be placed in the first ‘P’ level found. (See the “Databases” section in the MDS Concepts Manual for details on database levels.) TDB is case sensitive. The name you supply will be converted to uppercase before storing.

**date**

VMS Usage: `date_name`  

<table>
<thead>
<tr>
<th>type:</th>
<th>character string</th>
</tr>
</thead>
<tbody>
<tr>
<td>access:</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism:</td>
<td>by descriptor</td>
</tr>
</tbody>
</table>

Optionally, the date of the shot desired. The date defaults to that set by a TDB$SET_DEFAULT call, or the current date if there was no preceding call to TDB$SET_DEFAULT. The format of the date is dd-mmm-year or the keyword “TODAY” for the current date or the keyword “LAST” for the date of the last shot taken.

**shot**

VMS Usage: `longword_signed`  

<table>
<thead>
<tr>
<th>type:</th>
<th>longword (signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access:</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism:</td>
<td>by descriptor</td>
</tr>
</tbody>
</table>

Optionally, the address of a longword containing the shot number desired. The shot number defaults to the shot set by a TDB$SET_DEFAULT call, or to the current shot if there was no preceding call to TDB$SET_DEFAULT.
Run-Time Library Routines
TDB$PUT_DATA

buflen
VMS Usage: longword_signed
Type: longword (signed)
Access: read only
Mechanism: by descriptor
The address of a longword containing the length in real*4's of the buffer containing the data to be stored.

buffer
VMS Usage: vector_floating_point
Type: F_floating
Access: read only
Mechanism: by reference
The address of a real*4 variable or array containing the data. The buffer must be dimensioned to at least buflen.

units
VMS Usage: mds_units
Type: character string
Access: read only
Mechanism: by reference
An optional string descriptor containing the units of the data. Units can be up to 8 characters long.

DESCRIPTION
TDB$PUT_DATA will store a processed data record in the selected MDS database. You must make a call to TDB$SET_DB to establish the current database prior to calling TDB$PUT_DATA. Unless a level prefix is explicitly given in the item name, the item will be placed in the 'P' level.

The error handling for this routine is selectable via the TDB$SET_ERRORS routine. Errors can result in brief error messages, full error messages, return status only, or a signal to a condition handler.

CONDITION VALUES

RETURNED

RMS$_NORMAL
The data has been successfully stored.

TDB$_PUTERR
There was an error writing the database item.

EXAMPLE

The following is an sample storage of the data "TF_CURRENT"

Real*4 CURRENT(8192)
CALL TDB$SET_DB('ALCATOR$')
CALL TDB$PUT_DATA('TF_CURRENT', CURRENT, 8192, amps')
END

! Declare the CURRENT array
! Set database to "ALCATOR$
! Store the trace of TF_CURRENT
! for the current date and shot
! End
TDB$PUT_LOGICAL

Write logical transformation item into an MDS database.

**FORMAT**

```
TDB$PUT_LOGICAL logname, [date], [shot],
equivname [,start [,end [,inc
[,eqdate [,eqshot]]]]]
```

**RETURNS**

VMS Usage: `cond_value`
type: `longword (unsigned)`
access: `write only`
mechanism: `by value`

**ARGUMENTS**

**logname**

VMS Usage: `mds_db_item_name`
type: `character string`
access: `read only`
mechanism: `by descriptor`
The name of the database item to be created. Database access is case sensitive. The name you supply will be converted to uppercase before insertion into the database.

The name may be prefixed by the database level specification, (for example: "S.MYITEM", where “S” is the level prefix). If a prefix is included, MDS will insert the item into the level in the database which has a matching prefix. Otherwise, the database will be placed in the ‘P’ level. (See the “Databases” section in the MDS Concepts Manual for details on database levels.)

**date**

VMS Usage: `date_name`
type: `character string`
access: `read only`
mechanism: `by descriptor`
Optionally, the date of the shot for which the transform is to apply. Defaults to the date set by TDB$SET_DEFAULT call or the current date if no call to TDB$SET_DEFAULT has been made. The format of the date is dd-mmm-year or the keyword “TODAY” for the current date or the keyword “LAST” for the date of the last shot taken.
Run-Time Library Routines
TDB$PUT_LOGICAL

**shot**
- VMS Usage: `longword_signed`
- Type: `longword (signed)`
- Access: `read only`
- Mechanism: `by reference`

Optionally, the address of a longword containing the shot number for which the transform is to apply. Defaults to the shot set by TDB$SET_DEFAULT call or the current shot if no call to TDB$SET_DEFAULT has been made. A value of -1 will use the current shot number.

**equivname**
- VMS Usage: `mds_db_item_name`
- Type: `character string`
- Access: `read only`
- Mechanism: `by descriptor`

The address of a character string descriptor containing the name of the record to which this logical transform is to reference. It may contain a level prefix but the entire name cannot exceed 23 characters.

**start**
- VMS Usage: `longword_signed`
- Type: `longword (signed)`
- Access: `read only`
- Mechanism: `by reference`

Optionally, the address of a longword containing the index of the first sample in `equivname`'s record you wish to receive when retrieving `logname`. If this argument is omitted or set to 0, retrieval will begin with the first sample in the record.

**end**
- VMS Usage: `longword_signed`
- Type: `longword (signed)`
- Access: `read only`
- Mechanism: `by reference`

Optionally, the address of a longword containing the index of the last sample in the `equivname`'s record you wish to receive. If this argument is omitted or set to 0, retrieval will end with the last sample in the record.

**inc**
- VMS Usage: `longword_signed`
- Type: `longword (signed)`
- Access: `read only`
- Mechanism: `by reference`

Optionally, the address of a longword containing the increment between samples in the record you wish to receive. For example setting `start` to 1, `end` to 100, and `inc` to 10 would retrieve samples 1,11,21,31,41,51,61,71,81, and 91. If this argument is omitted or set to 0 and increment of 1 will be used.
eqdate
VMS Usage: date_name
type: character string
access: read only
mechanism: by descriptor
An optional string descriptor to containing the date of the record to point to.

act-shot
VMS Usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference
Optionally, the address of a longword to containing the shot number of the record to point to.

DESCRIPTION
TDB$PUT_LOGICAL will store a logical transform in the TDB database. You must make a call to TDB$SET_DB to establish the current database prior to calling TDB$PUT_LOGICAL.

CONDITION VALUES RETURNED
RMS$NORMAL
The item has been successfully stored.
TDB$PUTERR
There was an error putting the record into the data file.

EXAMPLE
The following is an sample storage of the transform "TF_CURRENT":

Real*4 CURRENT(8192)
CALL TDB$SET_DB('ALCATOR$')
CALL TDB$PUT_LOGICAL('TF_CURRENT',
+ 'SIGNAL_1035')
END

! Declare the CURRENT array
! Set database to "ALCATOR$"
! Store the transform TF_CURRENT
! for the current date and shot
! End
TDB$RAWDATA

Get MDS database item (unprocessed).

**FORMAT**

```
TDB$RAWDATA name, [date], [shot], buflen,
   buffer [,bytcnt [,units [,source
   [,act-date [,act-shot [,dtype]]]]]]
```

**ARGUMENTS**

- **name**
  - VMS Usage: `mds_db_item_name`
  - Type: `character string`
  - Access: `read only`
  - Mechanism: `by descriptor`
  - The name of the database item desired. Database access is case sensitive. The name you supply must match exactly the name in the database for retrieval to take place. In almost all cases the database item name will contain only upper case letters.
  - The name may be prefixed by the database level specification, (for example: “S.MY_ITEM”, where “S.” is the level prefix). If a prefix is included, MDS will search only the levels in the database which have matching prefixes. Otherwise, all levels will be searched for the database item. (See the “Databases” section in the *MDS Concepts Manual* for details on database levels.)

- **date**
  - VMS Usage: `date_name`
  - Type: `character string`
  - Access: `read only`
  - Mechanism: `by descriptor`
  - Optionally, the date of the shot desired. If omitted, the date defaults to the date set by TDB$SET_DEFAULT call or the current date if no call to TDB$SET_DEFAULT has been made. The format of the date is dd-mmm-yyyy, or the keyword ”TODAY” for the current date, or the keyword ”LAST” for the date of the last shot taken.

- **shot**
  - VMS Usage: `longword_signed`
  - Type: `longword (signed)`
  - Access: `read only`
  - Mechanism: `by reference`
  - Optionally, the address of a longword containing the shot number desired. If omitted, the shot defaults to the shot set by a TDB$SET_DEFAULT call or the current shot if no call to TDB$SET_DEFAULT has been made. A value of -1 will use the current shot number.
Run-Time Library Routines
TDB$RAWDATA

start
VMS Usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference
Optionally, the address of a longword containing the index of the first sample in the record you wish to receive. If this argument is omitted or set to 0, retrieval will begin with the first sample in the record.

end
VMS Usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference
Optionally, the address of a longword containing the index of the last sample in the record you wish to receive. If this argument is omitted or set to 0, retrieval will end with the last sample in the record.

inc
VMS Usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference
Optionally, the address of a longword containing the increment between samples in the record you wish to receive. For example setting start to 1, end to 100, and inc to 10 would retrieve samples 1,11,21,31,41,51,61,71,81, and 91. If this argument is omitted or set to 0 an increment of 1 will be used.

buflen
VMS Usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference
The address of a longword containing the length in bytes of the buffer into which the data is to be returned.

buffer
VMS Usage: vector_byte_unsigned
type: one of the MDS data types
access: write only
mechanism: by reference
The address of a buffer to receive the data. The buffer must be least buflen bytes in length.
bytcnt
VMS Usage: longword_signed
   longword (signed)
type:  write only
access:  by reference
Optionally, the address of a longword to receive the number of bytes returned
in the buffer.

units
VMS Usage: mds_units
   character string
type:  write only
access:  by descriptor
An optional string descriptor to receive the units of the data. Units can be
up to 8 characters long.

source
VMS Usage: mds_source
   character string
type:  write only
access:  by descriptor
An optional string descriptor to receive the source of the data. The source
of a TDB transform is the username of the user that created the transform.
The source of a digitizer data record is the CSV module name which created
the record.

act-date
VMS Usage: date_name
   character string
type:  write only
access:  by descriptor
An optional string descriptor to receive the actual date of the TDB record. For
shot data this date will be the same as that requested. For TDB transforms
this date represents the first day that the transform applies.

act-shot
VMS Usage: longword_unsigned
   longword (unsigned)
type:  write only
access:  by reference
Optionally, the address of a longword to receive the actual shot number of
the TDB record. For shot data this shot number will be the same as that
requested. For TDB transforms this shot number represents the first shot
that occurred on act-date that the transform applies.
Run-Time Library Routines

TDB$RAWDATA

dtype
VMS Usage: byte unsigned
type: byte unsigned
access: write only
mechanism: by reference
An optional byte to receive the data type of the record. (See the “Supported Data Types” appendix in the MDS Concepts Manual for a summary of the MDS data types.)

DESCRIPTION
TDB$RAWDATA will return a specified data record in its original format. You must make a call to TDB$SET_DB to establish the current database prior to calling TDB$RAWDATA. Only records such as digitizer data could be useful in this form. Transforms are not executed and if you retrieve a transform you just get back the definition record of that transform. For these reasons, the use of TDB$RAWDATA is discouraged.

CONDITION

VALUES

RETURNED

TDB$_NORMAL
The data has been successfully returned.

TDB$_TRUNCATED
Part of the data has been successfully returned but the buffer length was not large enough to receive all of the data. Informational.

TDB$_INVDB
No database was successfully selected using a TDB$SET_DB call.

SS$_IVTIME
The date supplied is invalid.

TDB$_RNF
The item was not found or not available.

EXAMPLE
The following is an sample retrieval on the signal “TF_CURRENT”:

```
Integer*2 CURRENT(8192)
CALL TDB$SET_DB('ALCDATA$')
CALL TDB$RAWDATA('TF_CURRENT,,16384,CURRENT')
END
```

MDSRTL 7-99
TDB$SET_DB

Select an MDS database for subsequent access.

**FORMAT**

TDB$SET_DB database

**RETURNS**

VMS Usage: cond_value

type: longword (unsigned)

access: write only

mechanism: by value

**ARGUMENTS**

database

VMS Usage: file_spec

type: character string

access: read only

mechanism: by descriptor

The address of a character string descriptor containing the name of the MDS database description file specification. If you omit the directory, the default [000000] is used. If you omit the file name, MDS is used, and if the type is omitted, .DATABASE is used. The database name can also be a logical name pointing to the database description file.

**DESCRIPTION**

The TDB$SET_DB routine reads the information in a database description file and establishes the context for subsequent database accesses. You must call this routine before attempting to access any MDS data.

See the “DATABASE” section of the MDS Concepts Manual for more detailed information on database definition files.

**CONDITION VALUES RETURNED**

<table>
<thead>
<tr>
<th>TDB$_NORMAL</th>
<th>The database has been successfully selected.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDB$_DBERR</td>
<td>The DATABASE command in the database description file has a syntax error.</td>
</tr>
<tr>
<td>TDB$_DBERR</td>
<td>The DATABASE command in the database description file has a syntax error.</td>
</tr>
<tr>
<td>TDB$_SETDBERR</td>
<td>Unable to access the database description file or unable to open the shot identification file.</td>
</tr>
</tbody>
</table>
EXAMPLE

See the TDB$DATA routine documentation for an example of the use of TDB$SET_DB.
TDB$SET_DEFAULT

Sets the default date and shot identification for the image. Subsequent database accesses will use this data and shot if they are not explicitly specified in the database access request.

**FORMAT**

| TDB$SET_DEFAULT [date [,shot]] |

**RETURNS**

VMS Usage: cond_value
type: longword (unsigned)
access: write only
mechanism: by value

**ARGUMENTS**

date
VMS Usage: date_name
type: character string
access: read only
mechanism: by descriptor

A string descriptor containing a date in the format dd-mmm-yyyy or the keyword “TODAY” for the current date or “LAST” for the date of the last shot taken (date of current shot). If omitted the default date remains unchanged. Initially, before any TDB$SET_DEFAULT call, the default date is set to “LAST” denoting the date of the last shot taken.

shot
VMS Usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference

A longword containing the default shot number. If omitted the default shot number remains unchanged. If set to -1, the default will be the last shot archived (current shot). Initially, before any TDB$SET_DEFAULT call, the default shot is set to -1, denoting the current shot.

**DESCRIPTION**

The TDB$SET_DEFAULT routine selects a default date and/or shot to be used for subsequent TDB$DATA calls. The initial default is the date and shot of the last shot stored in the database.

**CONDITION VALUES RETURNED**

- **TDB$NORMAL**: The default was successfully set.
- **SSS_IVTIME**: An invalid date has been supplied.
TDB$INVDB

The date and or shot selected required association to an MDS database to be resolved. You must call TDB$SET_DB prior to this call.

EXAMPLE

The following line of FORTRAN code will set the default date to the 24th of July 1984 and the default shot number to 10.

CALL TDB$SET_DEFAULT('24-JUL-1984', 10)
TDB$SET_ERRORS

Select error reporting for MDS database access routines.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>TDB$SET_ERRORS flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETURNS</td>
<td>VMS Usage: cond_value longword (unsigned) type: write only access: by value</td>
</tr>
<tr>
<td>ARGUMENTS</td>
<td>flag longword_signed type: read only access: by reference mechanism: by reference</td>
</tr>
</tbody>
</table>

A logical variable indicating the level of error reporting to perform. The following table will describe the options available:

<table>
<thead>
<tr>
<th>flag</th>
<th>Error handling selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No message, return status value</td>
</tr>
<tr>
<td>2</td>
<td>Signal error</td>
</tr>
<tr>
<td>3</td>
<td>Report brief message, return status value</td>
</tr>
<tr>
<td>otherwise</td>
<td>Report full message, return status value</td>
</tr>
</tbody>
</table>

DESCRIPTION

The TDB$SET_ERRORS routine sets the error reporting mode for database access calls. If flag is set to 0, errors will cause the current database operation to abort and a failure status will be returned. If flag is set to 2, errors will be signaled and it is up to the caller to handle the signaled condition. If flag is set to 3, a brief error message will be output (not signaled), and a status value will be returned. All other settings of the flag will cause a full error message to be output and a status value returned to the caller.

CONDITION VALUES RETURNED

<table>
<thead>
<tr>
<th>TDB$NORMAL</th>
<th>The flag has been successfully loaded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_ACCVIO</td>
<td>It was impossible to read the flag variable.</td>
</tr>
</tbody>
</table>
EXAMPLE

The following call will enable error message reporting:

CALL TDB$SET_ERRORS(.TRUE.)
Run-Time Library Routines
TDB$SET_LAST_SHOT

TDB$SET_LAST_SHOT

Set the shot specification which represents the “LAST” or current shot recorded in the MDS database. Setting the last shot affects all users who are accessing the same database. This routine should not be confused with the TDB$SET_DEFAULT routine which selects the default shot for your image only.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>TDB$SET_LAST_SHOT date, shot</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETURNS</td>
<td>VMS Usage: cond_value</td>
</tr>
<tr>
<td></td>
<td>type: longword (unsigned)</td>
</tr>
<tr>
<td></td>
<td>access: write only</td>
</tr>
<tr>
<td></td>
<td>mechanism: by value</td>
</tr>
</tbody>
</table>

ARGUMENTS
date
VMS Usage: date_name
 type: character string
access: read only
mechanism: by descriptor
A string descriptor containing a date in the format dd-mmm-yyyy.

shot
VMS Usage: longword_signed
 type: longword (signed)
access: read only
mechanism: by reference
A longword containing the shot number. For MDS databases with date and shot shot identifications the shot number cannot be larger than 65534.

DESCRIPTION
The TDB$SET_LAST_SHOT routine sets the last date and shot in an MDS database. Normally this routine is called by a CSV scanner module or a program that is run at the beginning of the archival of a shot’s data. Note, do not confuse this routine with the TDB$SET_DEFAULT routine. The TDB$SET_LAST_SHOT routine writes the date and shot to the database which then becomes the default for all users that have set default to the last shot.

CONDITION VALUES RETURNED

| TDB$NORMAL | The last shot was successfully set. |
| SS$IVTIME  | An invalid date has been supplied.  |
TDB$INVDB

The date and or shot selected required association to an MDS database to be resolved. You must call TDB$SET_DB prior to this call.

EXAMPLE

The following example will increment the current shot number.

```
PROGRAM INCREMENT_SHOT  ! Program INCREMENT_SHOT
Character*11 DATE    ! Date character string variable
Integer*4    SHOT   ! Shot number variable
CALL TDB$SET_DB(MY_DATABASE) ! Select database
CALL TDB$DATE_SHOT(DATE,SHOT)  ! Get current shot
CALL TDB$SET_LAST_SHOT(DATE,SHOT+1) ! Increment the shot number
END       ! End
```
TDB$SET_MISSING

Select the value to be returned which indicates missing values.

**FORMAT**

| TDB$SET_MISSING | missing |

**RETURNS**

| VMS Usage: | cond_value |
| type: | longword (unsigned) |
| access: | write only |
| mechanism: | by value |

**ARGUMENTS**

| missing |

| VMS Usage: | floating_point |
| type: | f_floating |
| access: | read only |
| mechanism: | by reference |

A real4 variable containing the value to use for missing values.

**DESCRIPTION**

The TDB$SET_MISSING routine sets the value to use for missing values. When performing transforms it is possible to have values in vectors which cause illegal operations. One good example is dividing by a vector that contains some zero values in it. Instead of invalidating the entire vector operation, MDS will fill in a missing value for results of invalid operations. Any further operation with a missing value also results in a missing value. When the result is returned to the user via a TDB$DATA call, the missing values are set to the value selected by this routine.

**CONDITION VALUES RETURNED**

| TDB$NORMAL | The flag has been successfully loaded. |
| SS$_ACCVIO | It was impossible to read the flag variable. |

**EXAMPLE**

The following call will set the missing value to 98765.00

```
CALL TDB$SET_MISSING(98765.00)
```
TDB$SFREE1_DD

Free one dynamic TDB descriptor.

**FORMAT**

| TDB$SFREE1_DD tdb-desc |

**RETURNS**

VMS Usage: cond_value  
type: longword (unsigned)  
access: write only  
mechanism: by value

**ARGUMENTS**

tdb-desc  
VMS Usage: tdb_descriptor  
type: longword (unsigned)  
access: modify  
mechanism: by descriptor

The address of a dynamic TDB descriptor. See the TDB$SGET1_DD routine description for details on a dynamic TDB descriptor.

**DESCRIPTION**

A call to TDB$SFREE1_DD will free the virtual memory associated with a dynamic TDB descriptor. The LENGTH and POINTER fields of the descriptor will be zeroed. The CLASS field is checked before freeing and must be TDB$K_CLASS_D or an error is returned. Dynamic TDB descriptors are created by calling TDB$SGET1_DD and some database access routines return database items as dynamic TDB descriptors.

**CONDITION VALUES RETURNED**

| SS$_NORMAL | The descriptor has been successfully freed. |
| LIB$_BADBLOADR | The POINTER field of the descriptor has been corrupted. The virtual memory database may be corrupted for your process and further TDB accesses may perform unpredictably. Check for exceeding array bounds in your code. |
| LIB$_INVSTRDES | The TDB descriptor is not of the class TDB$K_CLASS_D. |
Run-Time Library Routines
TDB$SFREE1_DD

EXAMPLE

The following is an sample retrieval on the signal "TF_CURRENT":

Include 'SYS$LIBRARY:FORM$DEF($TDBDEF)'
! Define TDB_DESCR structure
Record /TDB_DESCR/ TDB
! Declare record for TDB descriptor
CALL TDB$SET_DB('ALCDATA$')
! Set database to "ALCDATA$"
CALL TDB$DATA_D('TF_CURRENT', ..., TDB)
! Get the trace of TF_CURRENT
! for the current date and shot
CALL MY_SUB(TDB.LENGTH, VAL(TDB.POINTER))
! Do something with the data
CALL TDB$SFREE1_DD(TDB)
! Free the dynamic memory
END
! End
TDB$SGET1_DD

Get one dynamic TDB descriptor.

**FORMAT**

| TDB$SGET1_DD | length, dtype, shotid, source, units, tdb-descr |

**RETURNS**

- **VMS Usage:** cond_value
- **type:** longword (unsigned)
- **access:** write only
- **mechanism:** by value

**ARGUMENTS**

- **length**
  - **VMS Usage:** longword unsigned
  - **type:** longword (signed)
  - **access:** read only
  - **mechanism:** by reference
  - The length in bytes of the virtual memory buffer to be allocated.

- **dtype**
  - **VMS Usage:** byte unsigned
  - **type:** byte (unsigned)
  - **access:** read only
  - **mechanism:** by reference
  - The data type of the buffer allocated.

- **shotid**
  - **VMS Usage:** mds_shotid
  - **type:** longword (unsigned)
  - **access:** read only
  - **mechanism:** by reference
  - The shot identification number as returned by TDB$SHOT_ID.

- **source**
  - **VMS Usage:** mds_source
  - **type:** character string
  - **access:** read only
  - **mechanism:** by descriptor
  - The source of the data.
Run-Time Library Routines
TDB$SGET1_DD

units
VMS Usage: mds_units
type: character string
access: read only
mechanism: by descriptor
The units of the data.

tdb-descr
VMS Usage: tdb_descriptor
type: longword (unsigned)
access: by descriptor
mechanism: The address of a TDB descriptor. The FORTRAN structure definition for a TDB descriptor is included in the $TDBDEF module in the text library, SYS$LIBRARY:FORMDSDEF.TLB. A TDB descriptor is structured as follows:

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENGTH</td>
<td>0</td>
</tr>
<tr>
<td>POINTER</td>
<td>4</td>
</tr>
<tr>
<td>SHOTID</td>
<td>8</td>
</tr>
<tr>
<td>reserved</td>
<td>12</td>
</tr>
<tr>
<td>FTYPE</td>
<td>CLASS</td>
</tr>
<tr>
<td>SOURCE</td>
<td>(23 bytes)</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>28</td>
<td>32</td>
</tr>
<tr>
<td>UNIT</td>
<td>40</td>
</tr>
<tr>
<td>DATE-TIME</td>
<td>(8 bytes)</td>
</tr>
<tr>
<td>44</td>
<td></td>
</tr>
<tr>
<td>DATE-TIME (cont)</td>
<td>48</td>
</tr>
</tbody>
</table>

DESCRIPTION
The TDB$SGET1_DD routine allocates virtual memory to the descriptor and fills in the appropriate fields of the descriptor. If the descriptor already points to virtual memory that memory will be freed if the length is different otherwise it will be reused. The CLASS field of the descriptor is filled with TDB$K_CLASS_D.

CONDITION VALUES RETURNED

| SS$_NORMAL      | The descriptor has been successfully allocated. |
| LIB$_BADBLOSIZ  | The length was less than or equal to zero. |
LIB$INSVIRMEM Insufficient virtual memory. The request required more dynamic memory than was available from the operating system.
TDB$SHOT_ID

Computes and encoded MDS shot identification number.

**FORMAT**

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDB$SHOT_ID</td>
<td>[date] , [shot] , shotid</td>
</tr>
</tbody>
</table>

**RETURNS**

<table>
<thead>
<tr>
<th>VMS Usage:</th>
<th>cond_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>type:</td>
<td>longword (unsigned)</td>
</tr>
<tr>
<td>access:</td>
<td>write only</td>
</tr>
<tr>
<td>mechanism:</td>
<td>by value</td>
</tr>
</tbody>
</table>

**ARGUMENTS**

<table>
<thead>
<tr>
<th>Argument</th>
<th>VMS Usage:</th>
<th>type:</th>
<th>access:</th>
<th>mechanism:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>date_name</td>
<td>character string</td>
<td>read only</td>
<td>by descriptor</td>
<td>Optionally, the address of a character string descriptor containing the data in either dd-mmm-yyyy format or the keywords “TODAY”, for the current date, or “LAST”, for the date of the last shot archived. If omitted, the default date will be used.</td>
</tr>
<tr>
<td>shot</td>
<td>longword_signed</td>
<td>longword (signed)</td>
<td>read only</td>
<td>by reference</td>
<td>Optionally, the address of an longword containing the shot number. If omitted, the default shot number will be used.</td>
</tr>
<tr>
<td>shotid</td>
<td>mds_shot_id</td>
<td>longword (unsigned)</td>
<td>write only</td>
<td>by reference</td>
<td>The address of a longword to receive the shot identification number.</td>
</tr>
</tbody>
</table>
TDB$SHOT_ID computes an internal shot identification from the date and shot specified. The shotid is a 32-bit encoding of the date and/or shot number. The encoding results in an ASCII sortable identifier which is part of the index key for multi-shot data files. The index key for multi-shot database files is constructed out of the name, segment, and shotid in such a manner which will cause the records to be automatically sorted first by increasing name, then by decreasing date and/or shot number, and then by increasing segment. The algorithm for computing a shotid is somewhat complex. For a shot only database the shotid is constructed by complementing the 32-bit shot number and reversing the order of the bytes in the result. For a date/shot database the shotid is constructed by loading the days since a base date (1-JAN-1980) into the most significant 16 bits and the shot number into the least significant 16 bits, complementing the result and then reversing the byte order.

TDB$NORMAL
The shotid has been successfully computed.

SS$IVTIME
The date supplied is not a valid date format, use dd-mmm-yyyy format.

TDB$INVDB
There has been no call to TDB$SET_DB to establish the current database. It is necessary to call select a database to determine whether the shot identification number will be encoded in shot only or date/shot format.
Run-Time Library Routines

TDB$WAIT

TDB$WAIT

Wait for asynchronous database put operations to complete.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>TDB$WAIT</th>
</tr>
</thead>
</table>
| RETURNS | VMS Usage: cond_value  
|         | type: longword (unsigned)  
|         | access: write only  
|         | mechanism: by value  |

ARGUMENTS  None.

DESCRIPTION  Writes to an MDS database using the TDB$PUT routine are done asynchronously. That is, the put routine returns control to the caller immediately even though the actual I/O to the database may not have yet completed. If it is necessary to wait for this I/O to complete before continuing in the application a call to TDB$WAIT will perform this synchronization for you. TDB$WAIT will return control to the caller only after all TDB put I/O operations have completed.

CONDITION VALUES RETURNED

| RMS$_NORMAL | The I/O operations have completed.  
| RMS$_ISI | Invalid stream identifier.  
| No successful TDB$PUT calls were performed prior to calling TDB$WAIT. |
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td></td>
<td>vii</td>
</tr>
<tr>
<td>1</td>
<td>Introduction to MDS Run-Time Library Routines</td>
<td>1-1</td>
</tr>
<tr>
<td>2</td>
<td>Documentation Format for Library Routines</td>
<td>2-1</td>
</tr>
<tr>
<td></td>
<td>2.1 Routine name</td>
<td>2-1</td>
</tr>
<tr>
<td></td>
<td>2.2 Routine overview</td>
<td>2-1</td>
</tr>
<tr>
<td></td>
<td>2.3 Format</td>
<td>2-1</td>
</tr>
<tr>
<td></td>
<td>2.4 Returns</td>
<td>2-1</td>
</tr>
<tr>
<td></td>
<td>2.5 Arguments</td>
<td>2-2</td>
</tr>
<tr>
<td></td>
<td>2.6 Description</td>
<td>2-2</td>
</tr>
<tr>
<td></td>
<td>2.7 Condition Values Returned</td>
<td>2-2</td>
</tr>
<tr>
<td></td>
<td>2.8 Condition Values Signaled</td>
<td>2-2</td>
</tr>
<tr>
<td></td>
<td>2.9 Example</td>
<td>2-2</td>
</tr>
<tr>
<td>3</td>
<td>Calling Run-Time Library Routines</td>
<td>3-1</td>
</tr>
<tr>
<td></td>
<td>3.1 Return Status</td>
<td>3-1</td>
</tr>
<tr>
<td></td>
<td>3.2 Argument Passing</td>
<td>3-1</td>
</tr>
<tr>
<td>4</td>
<td>Linking to the Run-time Library Routines</td>
<td>4-1</td>
</tr>
<tr>
<td>5</td>
<td>MDS Data Access</td>
<td>5-1</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Specifying MDS Data Items</td>
<td>5-1</td>
</tr>
<tr>
<td>5.2</td>
<td>Reading Data</td>
<td>5-3</td>
</tr>
<tr>
<td>5.3</td>
<td>Writing Data</td>
<td>5-6</td>
</tr>
<tr>
<td>6</td>
<td>MDS Event System</td>
<td>6-1</td>
</tr>
<tr>
<td>6.1</td>
<td>Issuing MDS Events</td>
<td>6-1</td>
</tr>
<tr>
<td>6.2</td>
<td>Waiting for an MDS Event</td>
<td>6-2</td>
</tr>
<tr>
<td>6.3</td>
<td>Queuing MDS Events</td>
<td>6-2</td>
</tr>
<tr>
<td>6.4</td>
<td>Establishing an MDS Event AST</td>
<td>6-4</td>
</tr>
</tbody>
</table>
### Section 7  Run-Time Library Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCL$DO_COMMAND</td>
<td>7-2</td>
</tr>
<tr>
<td>CSV$ASSIGN</td>
<td>7-4</td>
</tr>
<tr>
<td>CSV$CAMCHK</td>
<td>7-6</td>
</tr>
<tr>
<td>CSV$DO_COMMAND</td>
<td>7-8</td>
</tr>
<tr>
<td>CSV$LAM_SUPPORT</td>
<td>7-10</td>
</tr>
<tr>
<td>CSV$$PUT_CHANNEL</td>
<td>7-12</td>
</tr>
<tr>
<td>CSV$$PUT_CHANNEL2</td>
<td>7-16</td>
</tr>
<tr>
<td>CSV$$PUT_CLOCK</td>
<td>7-20</td>
</tr>
<tr>
<td>CSV$$PUT_TIMING</td>
<td>7-23</td>
</tr>
<tr>
<td>CSV$$PUT_VECTOR</td>
<td>7-27</td>
</tr>
<tr>
<td>MDS$DEQ_ALL</td>
<td>7-29</td>
</tr>
<tr>
<td>MDS$DEQ_EVENT</td>
<td>7-31</td>
</tr>
<tr>
<td>MDS$EVENT</td>
<td>7-33</td>
</tr>
<tr>
<td>MDS$EVENTAST</td>
<td>7-35</td>
</tr>
<tr>
<td>MDS$EVENTCAN</td>
<td>7-37</td>
</tr>
<tr>
<td>MDS$GETEVI</td>
<td>7-39</td>
</tr>
<tr>
<td>MDS$ISTHERE</td>
<td>7-41</td>
</tr>
<tr>
<td>MDS$Q_EVENT</td>
<td>7-43</td>
</tr>
<tr>
<td>MDS$FREE1_DD</td>
<td>7-45</td>
</tr>
<tr>
<td>MDS$GET1_DD</td>
<td>7-47</td>
</tr>
<tr>
<td>MDS$WTEVENT</td>
<td>7-49</td>
</tr>
<tr>
<td>PCL$DO_COMMAND</td>
<td>7-51</td>
</tr>
<tr>
<td>TDB$ASCTIM</td>
<td>7-53</td>
</tr>
<tr>
<td>TDB$DATA</td>
<td>7-55</td>
</tr>
<tr>
<td>TDB$DATA_D</td>
<td>7-60</td>
</tr>
<tr>
<td>TDB$DATE_SHOT</td>
<td>7-64</td>
</tr>
<tr>
<td>TDB$DO_COMMAND</td>
<td>7-66</td>
</tr>
<tr>
<td>TDB$EVENT_SHOT</td>
<td>7-68</td>
</tr>
<tr>
<td>TDB$FILE_NAME</td>
<td>7-70</td>
</tr>
<tr>
<td>TDB$FIND_RECORD</td>
<td>7-72</td>
</tr>
</tbody>
</table>
Contents

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDB$FINISH</td>
<td>7-76</td>
</tr>
<tr>
<td>TDB$GET</td>
<td>7-77</td>
</tr>
<tr>
<td>TDB$GETDBI</td>
<td>7-80</td>
</tr>
<tr>
<td>TDB$GET_SHOT</td>
<td>7-86</td>
</tr>
<tr>
<td>TDB$PUT</td>
<td>7-88</td>
</tr>
<tr>
<td>TDB$PUT_DATA</td>
<td>7-91</td>
</tr>
<tr>
<td>TDB$PUT_LOGICAL</td>
<td>7-93</td>
</tr>
<tr>
<td>TDB$RAWDATA</td>
<td>7-96</td>
</tr>
<tr>
<td>TDB$SET_DB</td>
<td>7-100</td>
</tr>
<tr>
<td>TDB$SET_DEFAULT</td>
<td>7-102</td>
</tr>
<tr>
<td>TDB$SET_ERRORS</td>
<td>7-104</td>
</tr>
<tr>
<td>TDB$SET_LAST_SHOT</td>
<td>7-106</td>
</tr>
<tr>
<td>TDB$SET_MISSING</td>
<td>7-108</td>
</tr>
<tr>
<td>TDB$FREE1_DD</td>
<td>7-109</td>
</tr>
<tr>
<td>TDB$GET1_DD</td>
<td>7-111</td>
</tr>
<tr>
<td>TDB$SHOT_ID</td>
<td>7-114</td>
</tr>
<tr>
<td>TDB$WAIT</td>
<td>7-116</td>
</tr>
</tbody>
</table>
Preface

Intended Audience

This Manual is intended for all users of the MDS data acquisition system who need to write application programs which access MDS databases or use the MDS event system.

Structure of This Document

This document describes the use of the callable interface to MDS. Following is a breakdown of the subject mater presented in this manual, according to section.

- Section 1 Introduction, the introduction contains a brief description of the MDS Run-Time Libraries and how they are used.

- Section 2 Documentation Format, this section describes the format used for documenting the individual routines.

- Section 3 Calling Run-Time Library Routines, this section presents guidelines on how to call the routines in user applications.

- Section 4 Linking to the Run-Time Library Routines, this section describes the procedure for linking to the libraries.

- Section 5 MDS Data Access, this section presents the various routines
used for reading and writing MDS database items. It includes actual code examples of MDS database access.

- Section 6 **MDS Event System**, this section describes the routines used for issuing and recognizing MDS events. Sample programs are included.

- Section 7 **Run-Time Library Routines**, this section contains detailed descriptions of the routines.